PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Méthodologie expérimentale pour évaluer les caractéristiques des plateformes graphiques avioniques
Auteur: Author:	Vincent Legault
Date:	2014
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Legault, V. (2014). Méthodologie expérimentale pour évaluer les caractéristiques des plateformes graphiques avioniques [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1546/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1546/
Directeurs de recherche: Advisors:	Guy Bois
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

MÉTHODOLOGIE EXPÉRIMENTALE POUR ÉVALUER LES CARACTÉRISTIQUES DES PLATEFORMES GRAPHIQUES AVIONIQUES

VINCENT LEGAULT

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE INFORMATIQUE)

AOÛT 2014

© Vincent Legault, 2014.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MÉTHODOLOGIE EXPÉRIMENTALE POUR ÉVALUER LES CARACTÉRISTIQUES DES PLATEFORMES GRAPHIQUES AVIONIQUES

présenté par : LEGAULT Vincent

en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées

a été dûment accepté par le jury d'examen constitué de :

Mme NICOLESCU Gabriela, Doct., présidente

M. BOIS Guy, Ph.D., membre et directeur de recherche

M. DAGENAIS Michel, Ph.D., membre

DÉDICACE

à Céline et Pierre

RÉSUMÉ

Dans un contexte où l'industrie aéronautique ne cesse d'intensifier le développement de nouvelles fonctionnalités visuellement attrayantes et où le temps de mise en marché doit être le plus court possible, l'étalonnage rapide des performances de traitement graphique dans un environnement avionique certifié devient un enjeu important.

Avec cet ouvrage nous voulons démontrer qu'il est possible de déployer une application graphique haute performance sur une plateforme avionique qui utilise des composants graphiques certifiés COTS. De plus, nous souhaitons apporter à la communauté aéronautique une méthodologie leur permettant d'identifier les éléments nécessaires à l'optimisation d'un système graphique en plus de leur fournir des outils servant à mesurer la complexité de ce type d'application et les ressources nécessaires pour dimensionner adéquatement un système graphique en fonction de ses besoins.

Actuellement, aucun outil de profilage des performances graphiques dédié aux architectures embarquées critiques n'a encore été proposé. Il nous est donc venu l'idée d'implémenter un banc de test spécialisé qui serait une contribution adaptée et efficace à cette problématique. Notre solution réside dans l'extraction de paramètres graphiques essentiels d'une application héritée pour pouvoir ensuite les utiliser dans une application de génération d'image 3D.

ABSTRACT

Within a context where the aviation industry intensifies the development of new visually appealing features and where time-to-market must be as short as possible, rapid graphics processing benchmarking in a certified avionics environment becomes an important issue.

With this work we intend to demonstrate that it is possible to deploy a high-performance graphics application on an avionics platform that uses certified graphical COTS components. Moreover, we would like to bring to the avionics community a methodology which will allow developers to identify the needed elements for graphics system optimisation and provide them tools that can measure the complexity of this type of application and measure the amount of resources to properly scale a graphics system according to their needs.

As far as we know, no graphics performance profiling tool dedicated to critical embedded architectures has been proposed. We thus had the idea of implementing a specialized benchmarking tool that would be an appropriate and effective solution to this problem. Our solution resides in the extraction of the key graphics specifications from an inherited application to use them afterwards in a 3D image generation application.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
COTS	Commercial Off-The-Shelf
FOV	Field Of View
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
GPS	Global Positioning System
GPU	Graphics Processing Unit
HMI	Hazardously Misleading Information
IFPS	Instantaneous Frames Per Second
IMA	Integrated modular avionics
OPC	OpenGL Performance Characterization
OS	Operating System
RTCA	Radio Technical Commission for Aeronautics
RTL	Register-Transfer Level
SBC	Single Board Computer
SDK	Software Development Kit
SoC	System on Chip
SPEC	Standard Performance Evaluation Corporation
SVS	Synthetic Vision Systems

LISTE DES ANNEXES

CHAPITRE 1
INTRODUCTION

Avec l'arrivée de matériel informatique de plus en plus performant, l'industrie aéronautique intensifie depuis la dernière décennie le développement de nouvelles fonctionnalités visuellement attrayantes telles que les planches de bord tout écran (*glass cockpit*) [1], la vision synthétique ou encore, les cartes de navigation tridimensionnelles [2]. Bien que ce type de fonctionnalité soit rependu dans un large éventail de produits comme les jeux vidéo ou les GPS, l'insertion de ces technologies est retardée pour les systèmes avioniques, qui doivent se soumettre à des normes de conception très strictes.

Parmi ces nouvelles fonctionnalités visuelles, nous retrouvons entre autres les systèmes de vision synthétique [3]. Ces derniers permettent aux pilotes d'aéronef de voir une représentation synthétique du milieu dans lequel ils évoluent. Par exemple, lorsqu'un pilote d'hélicoptère se retrouve aveuglé dans une tempête de sable –une situation communément appelée *brownout*– il pourrait utiliser un afficheur de son cockpit, un viseur tête haute [4] ou son casque et ainsi bénéficier d'une représentation 3D de son environnement de pilotage qui pallierait au manque d'indices visuels extérieurs. Cependant, une défaillance éventuelle d'un tel dispositif pourrait engendrer des conséquences catastrophiques, à savoir la perte de vies humaines et/ou de l'appareil. Il est donc impératif que la conception et que le développement de ce type de système répondent à un ensemble d'exigences de sécurité très strictes. Parmi ces exigences nous retrouvons essentiellement celles reliées à la technologie (respect des contraintes de temps et démonstration de sûreté de fonctionnement). Il faut aussi que le système soit commercialement viable pour que ce dispositif reste compétitif. Pourtant, ces systèmes sont si complexes qu'il n'est pas envisageable, à l'heure actuelle, de concevoir un système avionique ayant les mêmes fonctionnalités graphiques que le téléphone intelligent dans la poche du pilote. Il est donc nécessaire de repenser les méthodologies de conception actuelles pour permettre de concevoir un système complexe qui assurera sa mission tout en restant sécuritaire.

1.1 Problématique

En aéronautique, les moteurs, systèmes de bord, capteurs, actuateurs et autres composants avioniques sont dits « certifiables » lorsqu'ils ont passé une évaluation de conception et passé leurs tests de certification [5]. Au fil du temps, des standards pour développer ces pièces d'équipement ont été développés par les organismes régulateurs, de concert avec l'industrie. Ces lignes directrices couvrent tous les niveaux du développement des composants avioniques, incluant les systèmes mécaniques, électroniques, logiciels, les tests, ainsi que l'intégration de ces derniers. Les plus populaires sont développées par la *Radio Technical Commission for Aeronautics* (RTCA). La norme RTCA DO-178 [5] concerne le processus de développement logiciel en avionique. Elle couvre tous les aspects importants pour la certification d'équipement comportant une composante logicielle incluant la planification du projet, le développement, la validation et la vérification. La présence de plus en plus grande de circuits numériques sur mesure (*Application-Specific Integrated Circuit*, ASIC), reprogrammables (*Field-Programmable Gate Array*, FPGA) et graphiques (*Graphics Processing Unit*, GPU) dans les systèmes avioniques, a incité les autorités à créer la norme RTCA DO-254 qui s'adapte à cette nouvelle réalité. Cette ligne directrice est l'équivalent matériel du standard DO-178 et implique une vérification et une validation au niveau de la description synthétisée (RTL).

Les systèmes utilisant un engin graphique haute performance nécessitent des capacités de traitements graphiques substantiels que les processeurs généraux actuels, qui répondent aux normes avioniques, ne sont pas capables d'effectuer par manque de puissance [6]. Les compagnies aéronautiques ne sont pas intéressées par la création d'ASIC spécialisé pour effectuer du traitement graphique, car la validation de ce type de composant peut s'avérer très complexe, longue et coûteuse, vu la complexité du circuit à réaliser et l'effet des outils de conception (synthétiseur) sur la réalisation finale du circuit [5], sans permettre le niveau de performance des GPU commerciaux. Jusqu'à tout récemment, aucun produit commercial bon marché (*Commercial Off-The-Shelf*, COTS) ne s'adressait à ce marché. Certaines compagnies proposent maintenant des solutions abordables. Parmi celles-ci, on retrouve les firmes ALT Software et la montréalaise Presagis qui offrent des pilotes certifiés DO-178B pour quelques processeurs graphiques d'ancienne génération [2]. Ces GPU ont des architectures simples

(pipeline graphique fixe[1]) donc plus faciles à certifier, mais en contrepartie, ils sont aussi beaucoup moins performants que les GPU contemporains. À noter que la société ENSCO offre une alternative logicielle [7]. Leur produit génère sur un processeur un GPU virtuel certifiable DO-178. Cette virtualisation est bien évidemment moins performante que les capacités de traitement de ces homologues matériels dédiés et s'adresse plus au marché des applications avioniques graphiques simples comme l'écran de vol principal présenté à la Figure 1 (*Primary Flight Display*, PFD).

Figure 1 : Exemple d'écran de vol principal (©2001 Phoenix Simulation Software)

Le moteur graphique requis pour une application telle qu'un système de vision synthétique (*Synthetic Vision Systems*, SVS) est très gourmand en ressource de traitement graphique et nécessite donc l'utilisation d'un GPU. Comme lors de toute activité de développement logiciel, chaque fois qu'un concepteur conçoit ou calibre un système où il y a un partage de ressources, il se heurte de façon récurrente à la même question : le traitement (graphique) de mon application surchargera-t-il le processeur de la plateforme avionique cible? Jusqu'à présent, aucun outil ne

[1] Les API offert par les cartes graphiques à pipeline fixe offrent aux développeurs d'application graphique très peu de flexibilité. Comme son nom l'indique le pipeline fixe est statique et le programmeur ne peut modifier aucun des algorithmes de rendu graphique utilisés. Pour un pipeline programmable, certaines étapes du pipeline sont combinées (ou ajoutées) et remplacées par un programme (du code programmé par le développeur).

semble pouvoir aider les concepteurs à faire un choix éclairé à ce sujet dans le domaine spécifique de l'aéronautique. Lorsque l'on veut connaître les performances d'un GPU on parle souvent d'outil de type banc de test (*benchmark*). Les bancs de test pour GPU actuels sont inadaptés aux contraintes spécifiques de l'avionique. En effet, ils sont plutôt destinés aux cartes graphiques de nouvelle génération dotées d'un pipeline graphique programmable (avec *shaders*), qui sont conçues plus particulièrement selon les besoins de l'industrie des jeux vidéo. Or, non seulement ces outils sont conçus pour tester des cartes graphiques dans un contexte non-avionique, mais ils offrent aussi des métriques de performance qu'il est difficile d'utiliser de par la relative offuscation des informations propriétaires nécessaires à l'interprétation des résultats.

Les bancs de test graphiques existants, qu'ils soient commerciaux ou propriétaires, offrent des métriques « synthétiques » sans relation avec les réelles conditions d'utilisation normales. Ces métriques sont dépendantes d'un ensemble complexe d'interactions entre l'architecture interne du processeur, son organisation, les pilotes associés, plusieurs couches de librairies de traitement graphique, jusqu'à l'application de test.

Idéalement, l'application ciblée reste la meilleure source d'information utile pour estimer les ressources nécessaires à son exécution. C'est en exécutant des parties élémentaires de celle-ci dans un contexte réaliste, que les résultats sont prouvés étant les plus significatifs [8]. La littérature propose de reproduire des « parties spécifiques » de l'application devant être supportée et de les exécutées dans un contexte réaliste qui permet d'en extraire des mesures de performance qui suivent une rigueur scientifique de la quantification. L'exécution sur une plateforme cible de ces parties spécifiques fournit une mesure utilisable pour :

- estimer les ressources matérielles consommées par l'application de référence,
- mesurer l'effet que chaque fonctionnalité graphique a sur l'ensemble de l'application,
- comparer rigoureusement différentes solutions matérielles entre elles,
- inférer/prédire la consommation totale de l'application cible et
- estimer l'évolution dans le temps des performances afin d'en extrapoler une tendance.

Sachant ces lacunes, il nous est donc venu l'idée d'implémenter un banc de test spécialisé qui serait une contribution adaptée et efficace à cette problématique. Notre solution réside dans l'extraction de paramètres graphiques essentiels d'une application héritée pour pouvoir ensuite les utiliser dans une application de génération d'image 3D.

1.2 Objectifs

Le premier objectif de notre travail est de démontrer qu'il est possible de réaliser des applications graphiques hautes performances dans un contexte avionique, certifiable et à faible coût. Plus précisément, nous voulons démontrer qu'il est possible d'implémenter des applications telles qu'un système de vision synthétique ou des cartes de navigation tridimensionnelles sur des plateformes avioniques critiques en utilisant des composants commerciaux (COTS). Les grands constructeurs aéronautiques ont toujours été réticents à utiliser des applications graphiques 3D dans leurs aéronefs, car ils craignent que les coûts liés au développement de matériel graphique certifié soient trop onéreux [2]. En montrant qu'il existe des composants COTS certifiés capables de générer assez de puissance de traitement graphique pour ce genre d'application, cela donnera aux constructeurs une plus grande flexibilité par rapport au type d'application graphique qu'ils désirent intégrer dans le cockpit de leurs futurs aéronefs.

Le deuxième objectif est de développer un outil spécialisé permettant d'étalonner avec précision le niveau de traitement graphique requis par une application avionique graphique haute performance sur une plateforme ciblée. Plus précisément, nous voulons concevoir un banc de test graphique pour système avionique qui sera paramétrable, réutilisable et multiplateforme incluant le processeur, le GPU et le système d'exploitation d'une plateforme. Cet outil permettra d'aider les concepteurs d'applications graphiques avioniques dans leurs activités de mitigation des risques. Il permettra d'analyser plusieurs produits et de s'assurer de leurs capacités de traitement graphique ou encore de déterminer lequel des produits offre le rendement voulu, et ce, au meilleur prix. D'autre part, l'outil développé pourra aussi être utilisé dans un contexte d'estimation applicative. Il servira à établir le budget en ressources graphiques sur une architecture donnée. Par exemple, dans un contexte d'architecture avionique modulaire intégrée (*Integrated modular avionics*, IMA), où les applications se partagent les mêmes ressources, notre

outil permettrait d'estimer la consommation de ressources graphiques de chacune des applications et de vérifier qu'il est possible de les faire cohabiter sur une seule plateforme.

1.3 Méthodologie

Pour mener à bien ce projet, nous l'avons divisé en trois phases, soit l'élaboration, le développement et la validation.

1.3.1 Phase I : Élaboration

La phase d'élaboration consiste à recueillir toutes les informations nécessaires à la réalisation d'un modèle de l'application.

Nous avons en premier lieu procédé à une revue de la littérature et à une analyse approfondie des normes qui régissent la conception des applications en avionique puisqu'il est apparent que la rigueur de conception a un impact sur les choix architecturaux lors de la conception, donc sur les performances que l'on sera capable d'extraire. Nous avons été ainsi en mesure de déterminer les requis nécessaires à la certification d'un tel type de système. Ces spécifications nous ont permis de limiter nos recherches aux éléments matériels et logiciels qui peuvent être certifiés dans un environnement avionique.

En deuxième lieu, une recherche portant sur les applications graphiques avioniques existantes nous a permis de bien cerner les technologies dominantes de ce domaine. Cette étude plus circonscrite visait à récolter de l'information sur les composantes matérielles et logicielles utilisées dans le développement d'application graphique pour avionique. Par exemple, trouver les bibliothèques graphiques spécialisées pour les systèmes à sécurité critique ou encore, identifier les GPU qu'il est possible d'utiliser pour accélérer le traitement graphique.

Finalement, nous avons caractérisé une application graphique fournie par nos partenaires industriels, les entreprises montréalaises CMC Electronics Esterline et CAE. Cette caractérisation consistait à établir la liste des principaux éléments graphiques susceptibles d'affecter les performances de traitement graphique de l'application étudiée (ex. : quantité d'objets 3D, taille de la texture, etc.). Tous ces éléments ont été repris dans la conception de notre banc de test.

1.3.2 Phase II : Développement

La phase de développement consiste à concevoir les outils nécessaires à l'évaluation des performances d'un engin graphique.

Premièrement, nous avons implémenté une application graphique paramétrable, qui répliquait les éléments graphiques recensés dans l'application caractérisée. L'objectif était de soumettre la plateforme à un stress continu, qui permettait d'assurer un contrôle de façon progressive, répétable et cohérente. Pour extraire une mesure qui peut être réutilisée à des fins de comparaison, il était nécessaire de se fixer une référence.

Deuxièmement, nous avons transformé cette application en banc de test en lui ajoutant un module de mesure de performance incluant la génération de tests personnalisés, la prise de mesure via des sondes logicielles minimalement intrusives et la récupération de mesure de performance en sortie.

1.3.3 Phase III : Validation

Cette dernière phase vise à évaluer et valider l'utilité du banc de test que nous avons conçu. Dans un premier temps, nous avons comparé les mesures obtenues à l'aide de notre banc de test avec l'application de référence fournie par nos partenaires industriels. Dans un deuxième temps, nous avons lancé notre banc de test sur plusieurs plateformes afin de valider que les tendances de performances relevées étaient indépendantes des plateformes sur lesquelles le banc de test était lancé. Finalement, nous avons qualifié l'impact sur les performances d'un système qu'avait chacun des éléments graphiques d'une application avionique graphique.

1.4 Contribution

En réalisant les objectifs mentionnés à la section 1.2, nous voulons démontrer qu'il est possible de déployer une application graphique haute performance sur une plateforme avionique qui utilise des composants graphiques certifiés COTS. De plus, nous souhaitons apporter à la

communauté aéronautique une méthodologie leur permettant d'identifier les éléments nécessaires à l'optimisation d'un système graphique en plus de leur fournir des outils servant à mesurer la complexité de ce type d'application et les ressources nécessaires pour dimensionner adéquatement un système graphique en fonction de ses besoins.

1.5 Organisation du mémoire

Ce mémoire est divisé comme suit. Le Chapitre 2 présente une revue de la littérature de l'état de l'art. Nous ferons un survol des diverses technologies utilisées dans un contexte de développement d'application graphique avionique. Ensuite, nous présenterons au Chapitre 3 l'étude de cas que nous avons réalisée dans le cadre de ce projet. Cette étude porte sur un prototype d'application graphique destinée à être intégré à un cockpit d'aéronef. Au Chapitre 4, nous présenterons les éléments nécessaires à l'optimisation d'un système graphique ainsi que la méthodologie que nous avons suivie pour réaliser notre banc de test qui a pour objectif principal d'aider les concepteurs à mesurer la complexité de ce type d'application et des ressources nécessaires pour dimensionner adéquatement ces systèmes en fonction de leurs besoins. Après quoi, au Chapitre 5, nous dévoilerons les détails techniques de notre banc de test. Finalement, nous présenterons au Chapitre 6 les résultats que nous avons obtenus sur quelques architectures puis conclurons sur les améliorations futures de l'outil développé.

CHAPITRE 2
ÉTAT DE L'ART

2.1 Exemple d'une application avionique graphique

Comme mentionné plus tôt, grâce à l'essor du matériel informatique conforme aux standards avioniques, plusieurs constructeurs développent maintenant des instruments de bord avec un affichage sophistiqué. Parmi les projets les plus populaires, on retrouve les systèmes de vision synthétique. Ce dernier est un outil servant à générer une représentation synthétique tridimensionnelle de l'environnement dans lequel un aéronef est en vol, et ce, sans interférences par les conditions météorologiques, le niveau d'obscurité ou tous autres facteurs détériorant la visibilité d'un pilote [9]. Les environnements ainsi synthétisés sont affichés dans la cabine de pilotage pour que l'équipage puisse ensuite s'y référer afin de bénéficier d'une représentation équivalente au milieu survolé par l'aéronef. Cette représentation est généralement composée d'une reproduction en 3D du terrain, avec des symboles pour les pistes d'atterrissage et les obstacles au sol. Le tout est jumelé à de l'information 2D (symbologie) telle que les trajectoires de vol ou encore une série de données sur le statut de vol. La Figure 1 illustre bien ce genre de système.

Ces systèmes sont composés de trois éléments clés [10] : une base de données renfermant les données topographiques du terrain, un système de navigation permettant de déduire la position courante ainsi que l'orientation d'un aéronef et finalement, un système d'affichage temps-réel qui permet de générer et d'afficher un environnement de synthèse selon les données en provenance de la base de données et la situation de l'appareil dans cet environnement, basé sur les informations du système de navigation.

Figure 2 : Exemple d'application SVS (*SmartDeck*, conçu par Esterline CMC Electronics)

2.2 Cartes graphiques COTS certifiées

Pour réaliser un système SVS, les concepteurs font face à plusieurs défis de taille. Le plus important d'entre eux est probablement la difficulté d'optimiser l'application d'un environnement de développement commercial vers un environnement avionique [3]. Une des raisons pour laquelle il en est ainsi réside dans la nature du développement de logiciel graphique haute performance. En effet, ces applications gourmandes en traitement graphique seront déployées sur des plateformes restreintes en ressources, limitées en puissance d'alimentation, soumises à des températures extrêmes et tributaires d'une certification des plus strictes qui requière l'assurance de performance et de l'exécution conforme aux spécifications, en tout temps [11].

Dans les systèmes temps réel critiques, une attention particulière doit être accordée aux risques de malfonctionnement associés à une partie (ou l'ensemble) d'un système. Ces risques sont évalués au niveau de l'aéronef pour leurs impacts respectifs. Pour les systèmes d'affichage, les risques les plus communs sont les informations visuelles trompeuses (*Hazardously Misleading Information*, HMI) et l'arrêt de fonctionnement. Les HMI sont des informations

visuelles incorrectes sur un écran et l'arrêt de fonctionnement se traduit par une perte d'une partie ou de la totalité des données d'affichage (un écran noir par exemple).

Bien qu'il existe des solutions à base de FPGA et autres circuits intégrés (ASIC) pour lesquelles il est plus facile d'appréhender les caractéristiques, les applications telles que les SVS, les cartes de navigation en trois dimensions ou encore les *glass cockpit* requièrent une plus grande capacité de traitement que ces composants peuvent offrir [2]. Certains industriels surmontent cette limitation en implantant des processeurs graphiques COTS (COTS *Graphical Processors*, CGP) conçus à l'origine pour le marché grand public, sur leurs plateformes avioniques non critiques, en espérant qu'une fois la démonstration faite de la fiabilité de cette solution, il sera plus facile de les utiliser dans des systèmes plus critiques [12]. Pour intégrer des composantes CGP à une plateforme critique, il est nécessaire d'obtenir de la part du fabricant des données qui démontrent que le dispositif est adapté à un environnement avionique et qu'il répond aux exigences de fiabilité de l'industrie [13]. En complément des contraintes matérielles, les concepteurs doivent aussi prendre en considération d'autres points critiques lors de la sélection d'un CGP pour un système :

- s'assurer qu'il est possible d'avoir un soutien technique inconditionnel de la part du manufacturier du CGP tout au long de la phase de développement du produit,
- s'assurer que le manufacturier offrira du support pour le CGP pour au moins une décennie,
- puis confirmer la disponibilité de pilote graphique pour le CGP conçu pour supporter les systèmes d'exploitation temps réel et conforme aux normes de certification de sécurité critique internationales telles que le standard DO-178.

2.3 Le profil d'OpenGL SC

Les applications avioniques opèrent généralement sur des systèmes d'exploitation temps réel (*Real-time operating system*, RTOS) à sécurité critique comme VxWorks 653 de Wind River, Lynx-OS-178 de LynuxWorks ou encore Integrity-178B de Green Hills. Puisque ce type de RTOS est destiné au marché niche de l'aviation, rares sont les fabricants de CGP qui développent

un pilote graphique supporté par ce type de RTOS [11]. Les pilotes graphiques utilisés dans ces applications critiques doivent être soigneusement conçus, et ce, en conformité avec les normes de sécurité critique reconnues par les autorités.

Suite à trois années d'effort, la communauté aéronautique a présenté à la conférence Siggraph 2005 en Californie un API spécialement conçu pour les applications avioniques graphiques nommé OpenGL SC [14]. Cet API est un ensemble normalisé de classes, de méthodes et de fonctions servant de façade par laquelle les pilotes graphiques offrent leurs services à la couche logicielle du système. Les avantages de cet API sont nombreux :

- Premièrement, les constructeurs de systèmes avioniques maximisent leur nombre de fournisseurs de CGP en plus de s'assurer de la disponibilité d'un CGP nouvelle génération dans un futur temps, entièrement rétrocompatible[15].
- Deuxièmement, ceci rend la plateforme transparente aux yeux des développeurs système, car l'API rompt toute dépendance envers une architecture définie [15]. De plus, cette propriété rend le travail des développeurs modulaire et réutilisable dans une modernisation future du système.
- Troisièmement, OpenGL est omniprésent dans la sphère de l'infographie. Cette domination du marché offre l'avantage aux entreprises aéronautiques d'avoir accès à un bassin de ressources qualifiées [16]. Étant donné les nombreuses plateformes sur lesquelles est supportée la librairie graphique, une grande majorité des développeurs d'applications graphiques se sont, avec le temps, familiarisés avec OpenGL.
- Finalement, les API pour librairies graphiques du marché traditionnel des ordinateurs de bureau sont excessivement complexes et offrent beaucoup trop de fonctionnalités d'où l'intérêt d'avoir un API réduit à un sous-ensemble de fonctionnalités essentielles au domaine de l'avionique [17]. En effet, sachant qu'il y a un facteur de multiplication des coûts important lors d'une activité de développement logiciel suivant le standard DO-178 [15] et qu'un API plus léger implique une implémentation matérielle réduite (entrainant par conséquent une baisse de chaleur produite par le composant), les manufacturiers de CGP destinés au domaine de l'aviation ont tout intérêt à miser sur cet API pour garantir le succès de leurs produits.

C’est donc pour ces avantages qu’OpenGL SC s’est imposé comme le standard de librairie graphique du marché de l’informatique critique.

2.4 Benchmarks graphique contemporains inadaptés

À notre connaissance, aucun banc de test graphique dédié aux architectures embarquées critiques n’a encore été proposé à ce jour. D’autre part, les bancs de tests graphiques actuels sont inadaptés aux plateformes avioniques embarquées et visent plutôt le marché des jeux vidéo[18].

2.4.1 Basemark® ES 2.0

Considérons par exemple Basemark® ES 2.0 de la compagnie Rightware [19], une des suites de bancs de tests les plus populaires utilisées pour évaluer les GPU employés dans l’industrie des appareils mobiles. Les bancs de tests proposés par cette suite ne sont pas appropriés pour les plateformes avioniques pour plusieurs raisons.

Premièrement, le rendu graphique de cet outil utilise l’engin graphique Unity 4.0 qui utilise l’API OpenGL ES 2.0 qui est un API étoffé et complexe pour être un candidat raisonnable au développement complet d’une base de pilote certifiable pour plateforme avionique [15].

Deuxièmement, plusieurs méthodes d’éclairage sophistiquées sont testées. Parmi celles-ci nous retrouvons l’éclairage par pixel (*per-pixel lighting*) utilisé notamment pour créer les effets de placage de relief (*bump mapping*), les textures de lumière (*lightmap*) souvent utilisées pour réaliser des effets d’ombrage détaillé fixe, et les ombres portées par carte d'ombres (*shadow mapping*) pour créer des effets d’ombrage dynamiques. Bien que ces méthodes d’éclairage ajoutent une grande richesse visuelle au rendu final d’une image de synthèse, il reste à prouver l’utilité de celles-ci dans un contexte avionique où priment avant tout la détection d’obstacles et la précision du rendu de l’environnement synthétique.

Troisièmement, divers effets de traitement d’image, superflus à une application graphique avionique, sont aussi testés. Dans cet ensemble d’effets testés nous retrouvons, les effets maquettes (*tilt & shift*), les flous lumineux (*bloom*) ou encore les effets de halo lumineux (*anamorphic lens flares*). D’autre part, nous retrouvons certains tests pour les effets de matériaux

tels que l'interpolation de Phong et les effets de particules qui, encore une fois, jouent plus sur la qualité visuelle de l'image de synthèse que sur l'exactitude du modèle rendu.

Finalement, les résultats et métriques qu'offre ce banc de test (et bon nombre de ses alternatives) sont des notes de performance qui ne sont pas/peu documentées. Ces notes peuvent être pratiques lorsque l'on veut estimer la performance relative entre deux GPU, mais qu'en est-il lorsque nous voulons savoir si un GPU sera capable de supporter une application graphique définie? Plusieurs paramètres seraient à surveiller lorsque nous voulons porter une application graphique vers une plateforme avionique. Parmi ceux-ci nous retrouvons par exemple, le temps de chargement des données de la RAM générique vers la RAM GPU, le niveau de détail de l'image (nombre de polygones rendus par image), la taille des textures utilisées ou encore, le temps de traitement par image.

2.4.2 SPECviewperf

SPECviewperf est un banc de test portable, écrit en C, qui mesure les performances d'OpenGL. Ce dernier a été développé par le groupe OpenGL Performance Characterization (OPC) qui est associé à la Standard Performance Evaluation Corporation (SPEC), une organisation américaine à but non lucratif qui vise à produire, créer, maintenir et soutenir un ensemble normalisé d'outils de mesure de performance pour système informatique. SPEC est considéré comme une référence dans le secteur et est endossé par plusieurs grands joueurs tels que Microsoft, Intel, IBM, Apple ou encore, Sun Microsystems [20].

L'objectif de l'OPC est de fournir des méthodologies claires et sans ambigüité pour comparer les performances des implémentations d'OpenGL, peu importe le fabricant de composants, le système d'exploitation et l'environnement de fenêtrage [21]. SPECviewperf mesure les performances graphiques d'un système via l'API d'OpenGL en dessinant des modèles 3D définis par différentes primitives comme celles que l'on pourrait retrouver dans une application graphique réelle. Ce banc de test émule ce qu'une application graphique ferait comme traitement d'image et mesure les performances de traitement qui s'y rattachent. SPECviewperf produit ensuite des résultats exprimés en FPS. Avec les années, SPECviewperf s'est imposé

comme étant la référence en termes de banc de test pour mesurer les performances graphiques d'une plateforme cible.

Pour utiliser SPECviewperf, il suffit de fournir à cet outil un ensemble de données qui définit un ou plusieurs modèles 3D et une liste des différentes fonctionnalités graphiques (*rendering state*) que l'on souhaite tester. Avec ces paramètres d'entrées, l'outil produira une série d'images de synthèse pendant un temps prédéterminé avec des animations entre chacune d'elles. Chaque activité de rendu des images traversera l'ensemble des *rendering states* OpenGL spécifié en entrée. Finalement, SPECviewperf produira un rapport [22] qui inclut une mesure de performance exprimée en FPS et des informations sur le système en cours de test (incluant une liste des fonctionnalités graphiques testées, le temps mis pour construire les modèles 3D, ou encore un compte-rendu détaillé sur les primitives OpenGL utilisées dans le test).

La dernière version de l'outil SPECviewperf offre à la base 8 ensembles de données (modèles et scènes 3D) qui représentent grosso modo ce que pourrait contenir les applications graphiques contemporaines. Il est cependant possible de fournir son propre ensemble de données afin que le test exécuté par l'outil représente plus fidèlement une application graphique réelle ciblée par l'utilisateur (ce qui n'est pas possible avec le banc de test Basemark® ES 2.0 présenté précédemment). Avec SPECviewperf, il est également possible de tester uniquement les fonctionnalités graphiques de notre choix, ce qui est avantageux lorsque l'on veut effectuer des tests qui simulent une application graphique avec peu d'effet visuel, destinée au marché de l'aviation par exemple.

Malgré la grande diversité de résultats générés par l'outil (majoritairement liés aux caractéristiques graphiques du test), un problème majeur se présente lors de la présentation des résultats de performance. Seul le FPS moyen est mesuré. Celui-ci prend en considération l'ensemble des images générées lors de l'exécution du test. Avec cette seule métrique, il est donc impossible de distinguer les pires cas de scénario et ainsi vérifier que les contraintes de temps réel sont respectées dans un système. De plus, il est très fastidieux de visualiser l'ampleur des impacts qu'a chacune des fonctionnalités graphiques sur le système cible. En somme, l'utilité de cet outil réside surtout dans la comparaison des performances de traitement graphique offertes entre deux plateformes. D'autre part, seul l'API d'OpenGL est actuellement supporté par SPECviewperf excluant ainsi les profiles embarqué (ES) et *safety critical* (SC) de la librairie

graphique. Cette dernière caractéristique de l'outil implique qu'il ne peut pas être un candidat pour la mesure de performances du traitement graphique sur système embarqué.

L'étude de l'outil SPECviewperf nous a toutefois permis d'établir une liste exhaustive des fonctionnalités graphiques à analyser lors de l'étude de cas que nous avons réalisée dans le cadre de ce projet. Le prochain chapitre présente cette analyse.

CHAPITRE 3
ÉTUDE DE CAS : SYSTÈME DE VISION SYNTHÉTIQUE

Les performances d'un système peuvent être testées de plusieurs manières par les bancs de test. Dans la sphère des applications graphiques 3D, une des méthodes les plus efficaces est d'essayer de reproduire le comportement d'un système réel [8]. En utilisant cette méthode pour développer notre banc de test, nous avons pour avantage de considérer le système graphique de l'application comme un tout et nous obtenons donc des mesures de performance fiables et pertinentes.

Pour mener à bien ce travail, nous avons donc pris pour étude de cas le prototype d'un SVS développé par un de nos partenaires industriels (système présenté à la section 2.1). Nous avons retenu ce type d'application, car il représente très bien le genre d'application 3D haute performance voulu par le marché de l'aviation. La partie graphique de cette application est composée d'une scène complexe qui est munie d'un terrain, d'obstacles et de plusieurs effets de rendu tels que du brouillard ou une interpolation de Gouraud pour le lissage du terrain. Ces propriétés graphiques du SVS font de ce type d'application un candidat parfait à l'appellation « application 3D haute performance ». Étudions maintenant les caractéristiques graphiques d'un SVS.

3.1 Définition d'une tuile de terrain

Un SVS est composé d'une base de données qui referme les données topographiques du globe terrestre. Ces données sont référencées selon des coordonnées longitudinales et latitudinales. L'ensemble des points topologiques contenus dans une zone définie entre deux angles de longitude et de latitude représente une partie de terrain qu'on appelle plus communément une « tuile de terrain ». En d'autres termes, ces tuiles représentent des petites portions de terrain du globe terrestre. Les échantillons topographiques de cette base de données sont toujours pris à distance fixe selon la résolution topographique du modèle défini dans la base de données. Prenons par exemple une tuile représentant un terrain de 10 km par 10 km. Si nous

utilisons une résolution de 1 km en longitude et de 2 km en latitude, cela implique que nous retrouverons une coordonnée à chaque kilomètre sur l'axe longitudinal et chaque deux kilomètres sur l'axe latitudinal pour une somme totale de 50 coordonnées d'élévation sur notre tuile de 10 km par 10 km.

Dans un environnement de visualisation synthétique, les tuiles sont des objets 3D généralement de forme carrée sur le plan XZ et de hauteur variable sur l'axe des Y. Cette hauteur variable représente l'élévation du terrain en un point (ou coordonnée géographique) donné. Les valeurs d'élévation de terrain proviennent directement de la base de données mentionnée dans le paragraphe précédent. Cela implique que chaque objet 3D représentant une tuile de terrain est composé de plusieurs points à espacement fixe et que chacune de ces tuiles comporte le même nombre de points. En joignant les points contigus d'une tuile selon un agencement triangulaire, une série de polygones est créée faisant place, au final, à un objet 3D décrivant un carré/tuile de terrain.

Une fois cet objet 3D créé, on applique ensuite une texture sur l'ensemble de la tuile et une couleur à chacun des échantillons topographiques. L'ajout de couleur s'inscrit dans une optique de facteur humain comme repère visuel pour aider le pilote à avoir le sens de sa position altimétrique par rapport au terrain. Cela sert, par exemple, à distinguer les étendues d'eau par rapport à la terre ferme : les lacs sont de couleur bleue et les plaines sont de couleur verte et les hautes élévations sont en blanc. Notons que les couleurs de chacune des coordonnées sont aussi définies dans la base de données. Un exemple de tuile de terrain est illustré à la Figure 3.

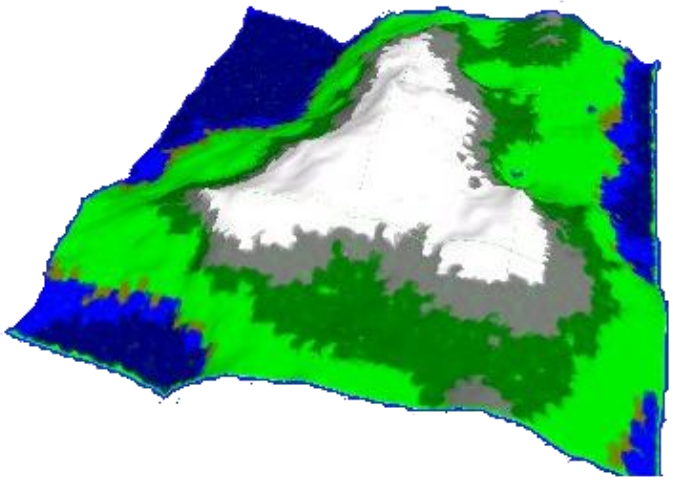

Figure 3 : Exemple de tuile de terrain (© www.synthetic-reality.com)

3.2 Définition d'une scène

Afin de former une scène riche et complète, plusieurs tuiles de terrain sont agencées côte à côte sous forme de damier. Dans le cas du prototype du SVS que nous avons caractérisé, les dimensions de ce damier sont de 31 tuiles par 31 tuiles. Les dimensions (en nombre de tuiles) de la scène doivent toujours correspondre à un carré composé d'un nombre impair de tuiles par coté, de sorte qu'une tuile centrale soit notre référence. La présence d'une tuile centrale est essentielle, car elle définit l'espace dans lequel le point de vue de l'observateur évolue. Cette propriété permet d'avoir en tout temps la même profondeur de scène, peu importe la direction visée par l'observateur. En effet, la position de l'observateur dans la scène est définie par la position géographique et l'altitude de l'aéronef piloté par ce dernier. De plus, la direction visée par l'observateur dépend de celle visée par ce même aéronef. Donc lorsque le pilote de l'appareil muni du SVS décide de changer de cap, l'horizon synthétisé a toujours une profondeur quasi constante.

Puisqu'il s'agit d'une application basée sur le géoréférencement d'un aéronef, la scène est dynamique dans le temps selon les coordonnées géographiques de l'appareil. Charger en mémoire vidéo la totalité des tuiles du globe terrestre est physiquement impossible, car cet ensemble de données est beaucoup trop volumineux. C'est pourquoi le SVS utilise une scène à dimension prédéfinie (réduite). Mais que se passe-t-il lorsqu'un pilote franchit les limites de la tuile centrale avec son aéronef? En fonction de la position de la nouvelle tuile survolée par l'appareil, les tuiles les plus éloignées « en arrière » sont effacées de la mémoire et de nouvelles tuiles « loin en avant » y sont chargées de sorte que la tuile au-dessus de laquelle le pilote se trouve devienne à son tour la tuile centrale. La Figure 4 illustre ce comportement avec un terrain constitué de 7X7 tuiles. Dans cet exemple, on peut voir un aéronef quitter la tuile centrale (orangée) vers la tuile nord-ouest (bleutée). Lorsque cet événement se produit, les tuiles grises sont éliminées de la mémoire, tandis que les tuiles vertes y sont chargées. On constate alors que du point de vue de l'observateur, il y aura toujours une profondeur de 3 tuiles de terrain pour effectuer le rendu graphique de l'environnement synthétique dans lequel il évolue.

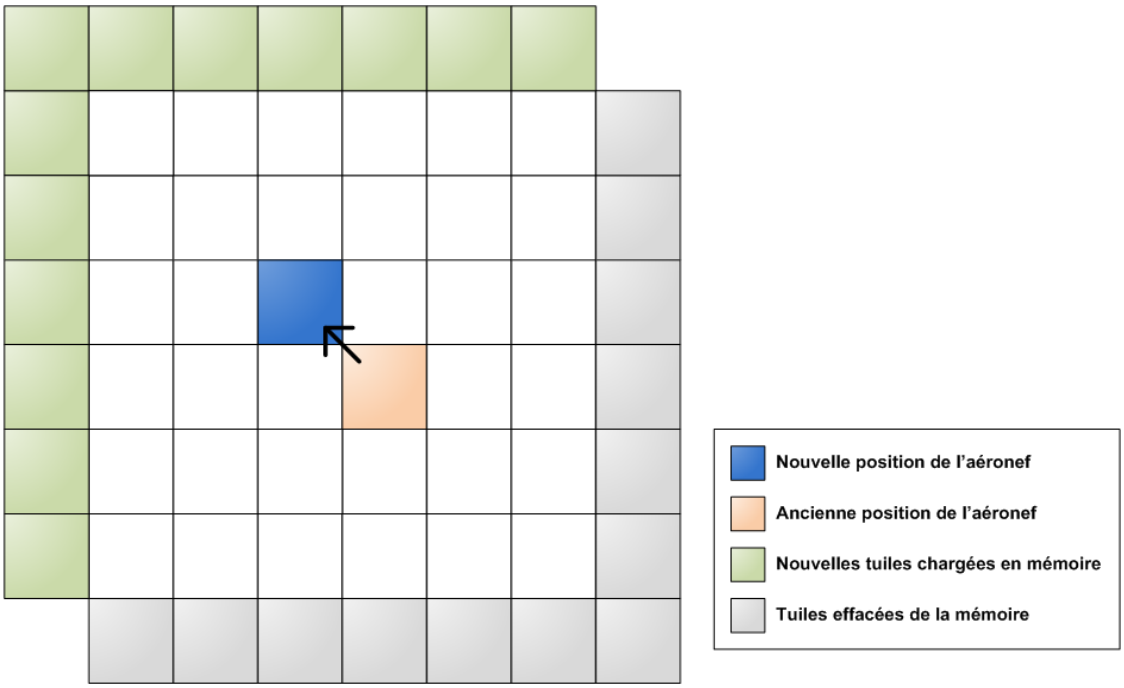

Figure 4 : Comportement du SVS lorsque l'aéronef quitte la tuile centrale

Pour optimiser les performances du traitement graphique, le prototype du SVS que nous avons étudié utilise deux niveaux de résolution pour afficher les tuiles de terrain. La logique derrière cette approche est qu'il n'est pas essentiel d'afficher en haute résolution les détails du sol (terrain) qui se trouve à une grande distance de l'aéronef (ou en d'autres termes : loin de la tuile centrale), car ces éléments offrent peu d'information au pilote quant à sa prise de décision à court terme. Il est tout de même important que les tuiles de terrain lointaines offrent assez d'information pour que le pilote puisse visualiser adéquatement l'endroit vers lequel il se dirige. En revanche, il est crucial pour le pilote de maximiser ses repères visuels ainsi que leurs niveaux de précision lorsqu'il se trouve plus près du sol. Les niveaux de résolution du prototype sont les suivants :

1. Haute résolution : tous les sommets de la tuile de terrain sont utilisées pour faire le rendu de celle-ci à l'écran;
2. Basse résolution : un sommet sur deux de la tuile est ignoré pour faire le rendu.

Les dimensions (en nombre de sommets) des tuiles doivent aussi être impaires afin de pouvoir joindre harmonieusement les tuiles de terrain limitrophes. En effet, si les tuiles étaient de dimension paire et que nous décidions d'afficher un sommet sur deux, la dernière ligne et la dernière colonne de sommets ne seraient pas affichées. Cela engendrerait un espace vide (crevasse) entre chaque tuile de terrain, une inconsistance visuelle non voulue. Notons que les sommets qui définissent les tuiles sont chargés intégralement en mémoire vidéo. Il est cependant possible d'avoir différents niveaux de résolution en omettant volontairement de rendre à l'écran certains sommets. Cette astuce permet d'accélérer le temps de traitement d'image en diminuant le niveau de détail du terrain synthétique. Ce mécanisme est rendu possible grâce à la double indexation des sommets. Nous en discuterons prochainement dans ce document.

Finalement, en plus de synthétiser le terrain, un SVS affiche aussi d'autres catégories de repère pour l'aide au pilotage. L'application étudiée affiche les pistes d'atterrissage ainsi que certains artéfacts humains (obstacles) qui sont eux aussi stockés dans la base de données. Dans le cas d'une piste d'atterrissage, on utilise une surface plane grisâtre sur laquelle on y inscrit le numéro d'identification de la piste et, dans le cas d'un obstacle, on utilise un prisme rectangulaire de couleur noire avec des dimensions proportionnelles à celles de l'obstacle à représenter. Ces éléments graphiques sont également représentés avec des objets graphiques 3D.

3.3 Rendu de la scène

Afin d'augmenter le réalisme 3D d'un environnement synthétique, il faut tenir compte de l'interaction entre la lumière et les surfaces de la scène. Plusieurs propriétés lumineuses doivent être spécifiées afin de remplir cette tâche. Nous nous concentrerons ici uniquement sur celles qui ont été utilisées dans le prototype du SVS que nous avons caractérisé. Nous nous limitons à ces propriétés, car ce que nous voulons implémenter est un banc de test qui reproduit le comportement d'une application réelle, soit le prototype SVS dans le cas présent. Les propriétés lumineuses se divisent en plusieurs catégories : les propriétés des matériaux, les sources de lumière et le modèle d'illumination.

Chaque surface d'objet 3D est composée d'une matière caractérisée par divers attributs. Ceux qui nous intéressent sont les réflectivités des lumières diffuse et ambiante :

1. Dans un contexte d'infographie, la lumière ambiante est présente partout dans l'environnement synthétique et n'a pas de direction[23]. Si ce type de réflexion est défini pour le matériel d'un objet, lorsqu'une lumière ambiante sera présente sur la scène, cela aura pour effet d'éclairer uniformément l'objet en question et ce peu importe la position de l'observateur sur la scène (position de la caméra synthétique).
2. Dans le prototype étudié, la lumière diffuse provient d'une source lumineuse directionnelle. Une lumière directionnelle illumine la totalité de la scène avec des rayons lumineux parallèles les uns aux autres. Ce type de lumière mime le comportement d'une source lumineuse infiniment éloignée comme le soleil par rapport à la Terre par exemple [23]. Lorsque ce type de lumière illumine un objet qui a la propriété matérielle de refléter la lumière diffuse, l'intensité de la réflexion dépendra directement de l'angle avec lequel la lumière frappe la surface de cet objet. La réflexion diffuse est plus brillante si les rayons lumineux atteignent directement une surface que s'ils l'effleurent. Lorsque la lumière touche une surface, son rayonnement est dispersé de manière égale dans toutes les directions en plus d'apparaitre également brillante quelle que soit la position de l'observateur. Cette propriété ne change pas la couleur d'une surface, mais plutôt l'intensité d'éclairage qu'elle reçoit. Elle est essentielle, car elle permet de bien visualiser les dénivellations à la surface d'un objet.

Pour augmenter la qualité du rendu à l'écran, le prototype utilise aussi certains effets graphiques. Nous retrouvons parmi ceux-ci le modèle d'illumination de Gouraud, un dispositif d'anticrénelage et des effets de brouillard pour flouter les frontières de la scène. D'autre part, afin d'optimiser la vitesse de rendu des images à l'écran, l'application masque les faces non visibles de la scène via une opération de *culling*. Autrement dit, cette procédure indique à l'engin graphique de faire abstraction de toutes les faces (triangles) situées derrière le terrain au premier plan, ce qui augmente la vitesse de traitement des images. Tous ces éléments seront discutés plus en détails à la section 5.3.2.

3.4 Informations 2D

En plus d'afficher de l'information 3D, le SVS mets aussi à la disposition des pilotes plusieurs autres informations en format 2D. Ces dernières peuvent être présentées sous forme de gauge ou de cadran, en caractères alphanumériques ou encore sous forme de symbologie propre au domaine de l'aéronautique comme un système de calcul de trajectoire *Highway-In-The-Sky* [24] par exemple. Dans un contexte graphique tridimensionnel tel que celui du SVS, l'affichage de ces éléments 2D peu complexes implique une consommation de ressource en traitement graphique négligeable. C'est pour cette raison que nous n'avons pas pris en considération ces éléments graphiques dans le cadre de la réalisation de notre projet de banc de test.

Dans le prochain chapitre, nous allons décrire la méthodologie que nous avons suivie pour réaliser notre banc de test. Cette méthodologie découle directement des observations faites dans le cadre de la caractérisation du prototype de l'application SVS

CHAPITRE 4
MÉTHODOLOGIE POUR LA RÉALISATION D'UN BANC DE TEST POUR APPLICATION GRAPHIQUE AVIONIQUE

Dans cette section, nous allons présenter la méthodologie que nous avons définie pour implémenter notre banc de test (nous nous y référerons pour le reste du document par le terme Outil). L'objectif ici est de réaliser un Outil qui mesure les performances de traitement graphique d'une architecture cible. Nous voulons offrir aux concepteurs d'application graphique avionique haute performance un banc de test leur permettant de calibrer leurs systèmes adéquatement de sorte qu'ils puissent tirer un maximum de profit des ressources matérielles présentes sur une plateforme définie.

4.1 Protocole utilisé

Puisque les architectures et les pilotes de GPU sont des propriétés intellectuelles gardées par les fabricants, notre banc de tests a été implémenté en utilisant le premier niveau d'abstraction atteignable (non restreint). Cette préférence pour une recherche de type empirique est imposée par le manque de ressources disponibles. Le premier niveau d'abstraction maitrisé est l'API de la librairie graphique. Cet API fournit une interface de communication entre les applications logicielles et les accélérateurs graphiques d'une plateforme. Cette façon de procéder implique que nous considérons les GPU comme des boites noires, c'est-à-dire que nous excitons le système en injectant des valeurs prédéfinies en entrée puis nous recueillons les résultats en sortie pour des fins d'analyse de performance. Ceci a pour désavantage principal de nous restreindre quant à la diversité des résultats des tests, mais à l'avantage de rendre l'Outil indépendant des architectures.

Nous avons choisi d'implémenter notre banc de test en C++, un langage de programmation largement utilisé dans le domaine aérospatial. Ce langage a notamment été adopté par le programme *Joint Strike Fighter* [25] qui est un programme américain de recherche aéronautique chargé de développer un avion de combat multi-rôle de nouvelle génération destiné à remplacer une grande partie des avions de chasse, de combat et d'attaque au sol en service dans les pays

sous influence de l'OTAN . Voici les caractéristiques qui on fait de ce langage de programmation un incontournable dans le domaine [26] :

1. Haut niveau d'abstraction;
2. Supporte la programmation orientée objet;
3. Sa portabilité et son efficacité : le langage est supporté sur la plupart des RTOS à sécurité critique (VxWorks 653, PikeOS ou encore Integrity-178B);
4. C'est un langage mature : il est bien documenté et il a été utilisé dans plusieurs types d'application critique;
5. Un large éventail d'outil de conception, d'analyse, de validation et de vérification est disponible pour ce langage;
6. La main-d'œuvre maitrisant ce langage est abondante.

Lockheed Martin, un des partenaires industriels de ce programme, a d'ailleurs instauré un guide de programmation C++ [27] destiné à aider les programmeurs à développer un code qui est conforme aux principes de sécurité des applications avioniques critiques, soit des applications épurées de code pouvant mener à des défaillances catastrophiques comme des blessures importantes aux membres d'un équipage et/ou des dommages à un aéronef.

Bien qu'il y ait une multitude de choix au niveau du langage de programmation pour implémenter la logique de notre application, il en est tout autrement pour le choix de la librairie graphique. Celui qui s'impose d'emblé est l'utilisation de la librairie graphique OpenGL ES [28]. En effet, cette librairie est un incontournable dans le marché de l'informatique embarqué, car elle est supportée par la plupart des manufacturiers de GPU et de RTOS. OpenGL ES définit un API allégé, bas-niveau et multiplateforme pour la conception d'applications générant des images 3D. Elle est une dérivée de la spécification de sa librairie mère OpenGL et est adaptée aux plateformes mobiles ou embarquées.

Les plateformes visées par l'Outil sont les plateformes avioniques à sécurité critique. Pourquoi ne pas utiliser la version « sécurité critique » d'OpenGL présentée à la section 2.3? Pour des raisons budgétaires et de rareté, nous n'avons pu mettre la main sur cette librairie pour

une architecture disponible. En effet, puisque tous les produits OpenGL SC sont faits en petite quantité, avec un soin particulier aux détails et avec des caractéristiques exclusives aux marchés militaires et avioniques, ceux-ci sont trop coûteux pour être utilisé dans les systèmes électroniques grand public [29]. Pour pallier à ce problème, nous avons plutôt opté pour la librairie graphique OpenGL ES 1.4. Grâce à la production et la consommation de masse des produits utilisant cette librairie graphique, spécialement depuis l'explosion du marché des téléphones intelligents, il est possible de mettre la main sur plusieurs GPU, OS et librairies compatibles OpenGL ES à des coûts très abordables. Cependant, puisqu'ultimement nous voulons que l'Outil soit exécuté sur un système utilisant la librairie graphique certifiée, nous avons utilisé uniquement les fonctions communes entre l'API d'OpenGL SC (présentés à la section 2.3) et celles d'OpenGL ES. En procédant de la sorte, nous nous laissons la possibilité de recompiler le code de notre outil sur une plateforme utilisant cette librairie.

Selon un expert en traitement graphique employé chez notre partenaire CMC Electronics Esterline, les librairies graphiques avec le profil « sécurité critique » sont généralement plus performantes que leurs homologues non certifiés, car elles comportent moins de fonctionnalités et que l'implémentation des fonctionnalités restantes est réalisée avec un nombre d'instructions réduit. L'expert soutient donc que cette implémentation moins complexe se traduit par une augmentation de performance de traitement graphique sur les systèmes utilisant ce type de librairie. Bien qu'il aurait été intéressant d'utiliser ce type de librairie dans notre étude, cela n'affecte pas la nature de nos expériences et l'utilité de l'Outil. En effet, notre banc de test sert à profiler les performances graphiques d'un système dans son ensemble, ceci incluant le CPU, le GPU, la RAM et la librairie graphique.

4.2 Conception du banc de test

Pour réaliser l'Outil, nous avons procédé en trois étapes. La première était de circonscrire les entrées du banc de test. La deuxième était de définir des métriques de sortie. Finalement, la dernière étape était d'imaginer un système qui allait nous permettre de générer des cas de test et qui allait organiser et classer les résultats recueillis en sortie.

4.2.1 Les entrées

Afin d'établir la liste des entrées de notre banc de test, nous nous sommes basés sur l'étude de cas du SVS présentée au Chapitre 3. Rappelons ici que pour réaliser un banc de test graphique qui produit des mesures de performance pertinentes, nous devons reproduire le comportement graphique d'une application réelle. Donc, la première étape de notre travail consistait à établir la liste de tous les paramètres pouvant affecter les performances de traitement graphique du système. En caractérisant le prototype du SVS, nous avons pu établir la liste de propriétés graphiques suivante :

1. Le nombre de sommets utilisés pour définir une tuile de terrain (résolution d'une tuile);
2. La taille de la texture utilisée sur une tuile;
3. Le nombre de tuiles utilisées pour constituer une scène;
4. Le modèle d'illumination : plat (*flat*) ou lissé (*smooth*);
5. L'absence ou la présence d'une fonctionnalité d'anticrénelage;
6. L'absence ou la présence d'effet de brouillard;
7. La dimension du tronc de projection[2];
8. La résolution de la fenêtre de projection.

La liste précédente correspond en fait aux paramètres d'entrées de notre banc de tests. L'objectif des tests réalisés par notre outil est de faire varier chacune de ces propriétés une à une et de mesurer leur impact sur les performances de traitement graphique d'une plateforme cible. Par exemple, en augmentant le nombre de sommets par tuile de test en test, nous pourrons mesurer le taux de dégradation des performances lié à la quantité de sommets utilisée pour

[2] En graphisme, un tronc de projection est la région tridimensionnelle qui est visible à l'écran. Ce volume a la forme d'une pyramide tronquée. Si on fait abstraction de cette troncature, le sommet de ce volume de forme pyramidal est la position de la caméra (point de vue de l'observateur). Le positionnement du tronc (inclinaison et rotation) dépend de l'endroit visé par la caméra. Pour plus de détails sur le tronc de projection, voir la section 5.3.1.2.

représenter une tuile dans une scène. Il s'agit maintenant de répéter cet exercice avec l'ensemble des propriétés énumérées dans la liste.

4.2.2 Les métriques de sortie

Comme nous l'avons mentionné un peu plus tôt, nous sommes restreints par rapport aux niveaux d'abstraction atteignable du système graphique d'une plateforme. Ceci a pour impact de nous limiter quant à la diversité des résultats qu'il nous est possible d'obtenir. En effet, puisque le premier niveau d'abstraction atteignable se situe au niveau de l'interface graphique logicielle, il nous est impossible d'obtenir des données situées au niveau matériel comme le taux d'utilisation du GPU, le taux d'occupation mémoire, le débit entre le CPU et le GPU, ou encore, l'impact des échanges graphiques sur le CPU. Ceci étant dit, les métriques de sortie pertinentes que nous pouvons obtenir sont le temps de traitement par image, le nombre de sommets présents dans le tronc de projection par image et le temps de chargement des données de la RAM système vers la RAM GPU.

L'unité de mesure la plus utilisée pour calculer les performances graphiques d'un système est le nombre d'images affichées par seconde (*Frame Per Second*, FPS). Elle est exprimée en images par seconde, elle est simple de compréhension et elle est utilisée depuis plusieurs années pour comparer les puissances de traitement entre les GPU [30]. Toutefois, ce type de mesure est loin d'être précis et, comme nous le verrons, il peut engendrer une mauvaise représentation de l'expérience utilisateur. Le calcul du FPS consiste à compter le nombre d'images rendues dans l'espace d'une seconde. Puisque l'être humain n'est pas assez rapide pour effectuer cette tâche, celle-ci est généralement déléguée à un outil de *benchmarking*, Basemark® ES 2.0 par exemple. Cet outil présenté à la section 2.4 utilise un FPS moyen, c'est-à-dire que le taux d'images par seconde est calculé sur une période de temps supérieur à 1 seconde. L'outil incrémente un compteur d'image en image sur une période de temps définie puis il calcule un résultat moyen final exprimé en images par seconde. Ce que cette méthode de mesure de performance ne nous montre pas est l'effet de « bond d'image » [31] dans les périodes ponctuelles où les unités de traitement graphique sont les plus sollicitées. Ce phénomène est illustré sur le graphique de la Figure 5. Ce dernier montre les temps de traitement par image pour deux applications. Bien que ces deux applications aient exactement le même FPS moyen (25 images par seconde), on peut

facilement constater que l'application B offre un temps de traitement par image non constant. Dans cet exemple, l'image #4 de l'application B prend 2 fois plus de temps que le FPS moyen pour être affichée. Dans un contexte d'application graphique temps réel, cette situation aurait des effets désagréables au niveau de la fluidité de mouvement de la caméra. Il est donc fondamental d'utiliser le *temps de traitement par image* pour bien identifier ces cas de scénario critiques. Pour faciliter la compréhension de ces mesures, nous avons décidé d'exprimer nos résultats en FPS instantané (*Instantaneous Frame Per Second*, IFPS) [32], soit l'inverse du temps de traitement par image.

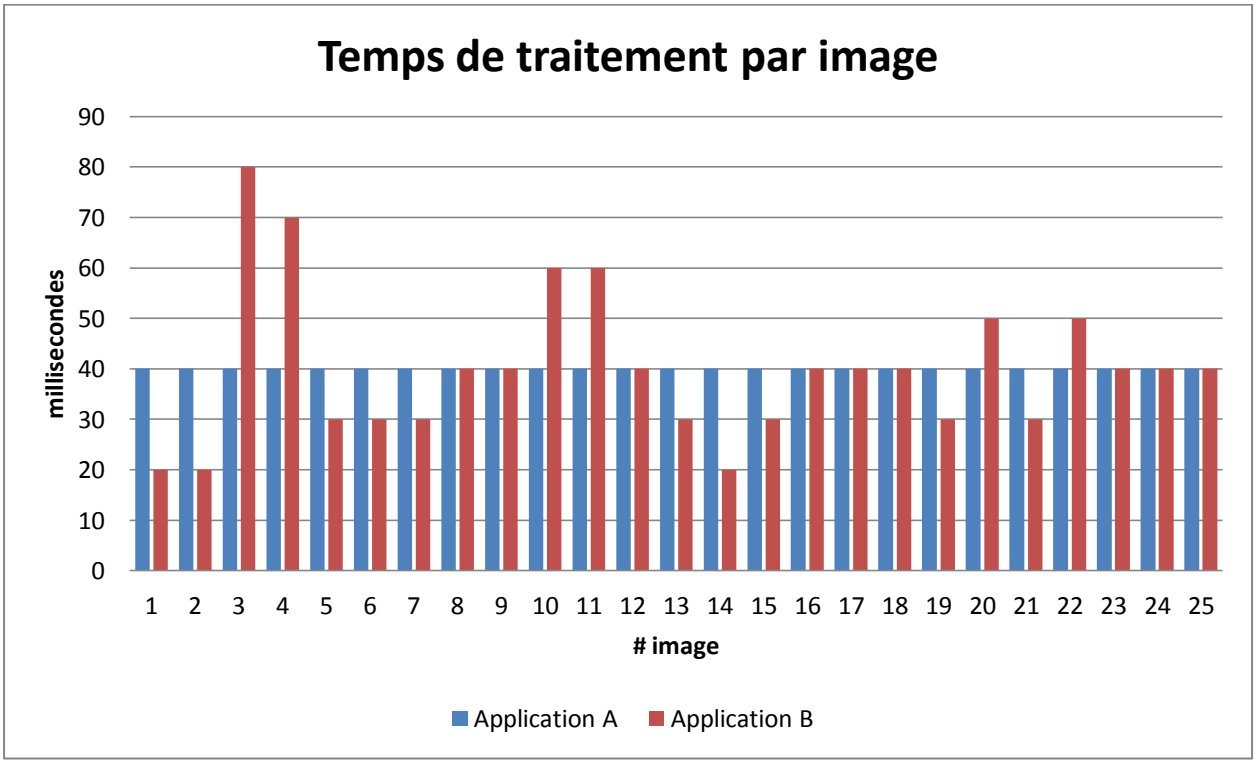

Figure 5 : Temps de traitement par image de deux applications graphiques

La deuxième mesure que nous prenons est le nombre de sommets présents dans le tronc de projection par image. Cette mesure sert à mesurer la dégradation des performances liée à la quantité d'information affichée à l'écran. Elle fournit pour chacune des images 3D générées le nombre de sommets qui ont été traités pour effectuer le rendu complet de l'image. Selon le point de vue de la caméra, ce nombre est plus petit ou égal au nombre de sommets des objets 3D

chargés en mémoire RAM GPU. Pour obtenir ces valeurs, nous calculons pour chacune des images générées le nombre de sommets présents dans le tronc de projection de la caméra.

4.2.3 Scénarios de test

Afin de rendre les résultats de l'Outil le plus répétable possible, nous avons dû imaginer des scénarios de prise de vue de caméra (point de vue de l'utilisateur) qui couvrent tous les cas d'utilisation. Quatre cas nous intéressent particulièrement :

1. Les prises de vue près du sol : seulement une petite quantité d'objets 3D (tuiles) sont rendus à l'écran.
2. Les prises de vue à distance moyenne du sol : moins de la moitié des objets 3D sont rendus à l'écran.
3. Les prises de vue à grande distance du sol : près de la totalité des objets 3D est rendue à l'écran.
4. Les prises de vue avec un champ de vision obstrué : un ou plusieurs objets 3D sont masqués (ou masqués en partie) par un ou des objets 3D situés dans l'axe de visualisation de la caméra.

Dans les quatre cas énumérés dans cette liste, la quantité d'objets 3D chargés en mémoire reste la même. Cependant, le nombre de polygones affichés à l'écran varie. Le premier objectif ici est de mesurer l'impact relatif au nombre de sommets inclus dans le tronc de projection sur les performances graphiques. En échantillonnant des prises de vue qui couvrent les cas #1, #2 et #3, nous pouvons extrapoler des résultats une courbe qui décrit la dégradation des performances liée au nombre de sommets inclus dans le tronc de projection. Le deuxième objectif vise à déterminer l'impact du tampon de profondeur (z-buffer). Cet objectif peut être atteint avec le cas d'utilisation #4. Ce que nous testons ici est l'incidence de ce tampon sur le FPS ou, en d'autres termes, nous voulons voir à quelle échelle la superposition d'objets 3D affecte la vitesse du traitement d'image.

Pour réaliser cet échantillonnage, nous avons dû procéder en deux temps. Premièrement, nous avons imaginé une scène paramétrable qui reproduisait fidèlement les pires scénarios en termes d'éléments graphiques à représenter sur une scène (toujours dans le contexte d'une application SVS). Ensuite, nous avons implémenté une caméra à positionnement automatisé permettant de générer des images de synthèse couvrant tous les angles de vues à différentes altitudes, selon les propriétés paramétrables de la scène, énumérées à la section 4.2.1.

La scène est constituée d'un damier de tuiles de formes pyramidales et irrégulières. Nous avons choisi des tuiles de forme pyramidale, car :

- cette forme reproduit un environnement semblable à une topographie synthétisée propre à notre application,
- cela permet de générer facilement des prises de vues avec un champ de vision obstrué (objectif #4 de nos scénarios de test). En effet, en prenant des prises de vues à basse altitude et en visant à l'horizontale, nous aurons un échantillonnage d'images ayant un fort taux de tuiles masquées par une ou plusieurs autres tuiles de terrain. Cette propriété est illustrée à la Figure 6.
- ce type de volume peut être facilement généré via une formule mathématique.

Pour éviter d'avoir des surfaces de tuile droites, nous avons ajouté un paramètre aléatoire permettant de générer des surfaces irrégulières, simulant une certaine ressemblance avec les variations de terrain que l'on trouve naturellement. La hauteur des tuiles est variable et dépend de la position de la tuile sur le damier. Les hauteurs sont exponentielles selon les index du diagramme illustré à la Figure 7. Plus le nombre est grand, plus la pyramide est haute. Les pyramides les plus petites ont une hauteur pratiquement nulle tandis que les plus hautes ont une hauteur totalisant le quart de la largeur de la scène (voir Figure 8). Ce choix de hauteur des tuiles a été retenu, car il permet de réaliser des prises de vue où le champ de vision est libre, partiellement obstrué par la dénivellation du terrain et totalement obstrué dans les cas extrêmes. D'autre part, nous avons choisi des hauteurs de pyramide extensibles et proportionnelles aux dimensions de la scène. Ceci permet d'éviter de générer des scènes « plates » lors du chargement

d’un très grand terrain (scène avec beaucoup de tuiles). On bénéficie ainsi d’un échantillonnage de hauteurs de pyramide qui respecte en tout temps les quatre objectifs de test.

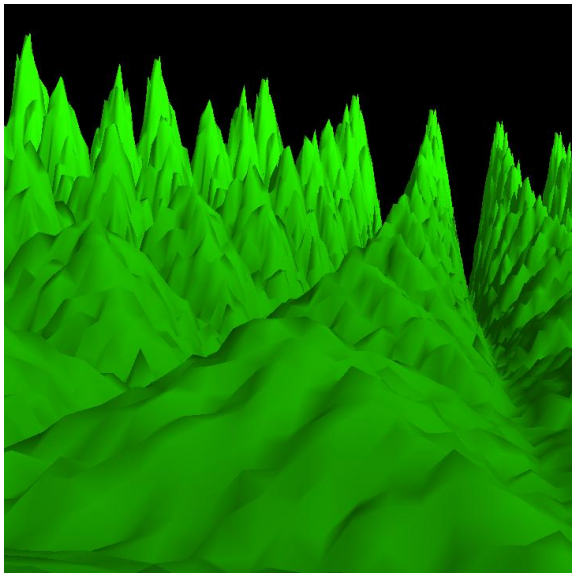

Figure 6 : Visualisation de la scène de test à basse altitude

0	1	2	3	4
3	0	1	2	3
2	3	0	1	2
1	2	3	0	1
0	1	2	3	0

Figure 7 : Hauteur des pyramides selon leurs positions sur la scène de test

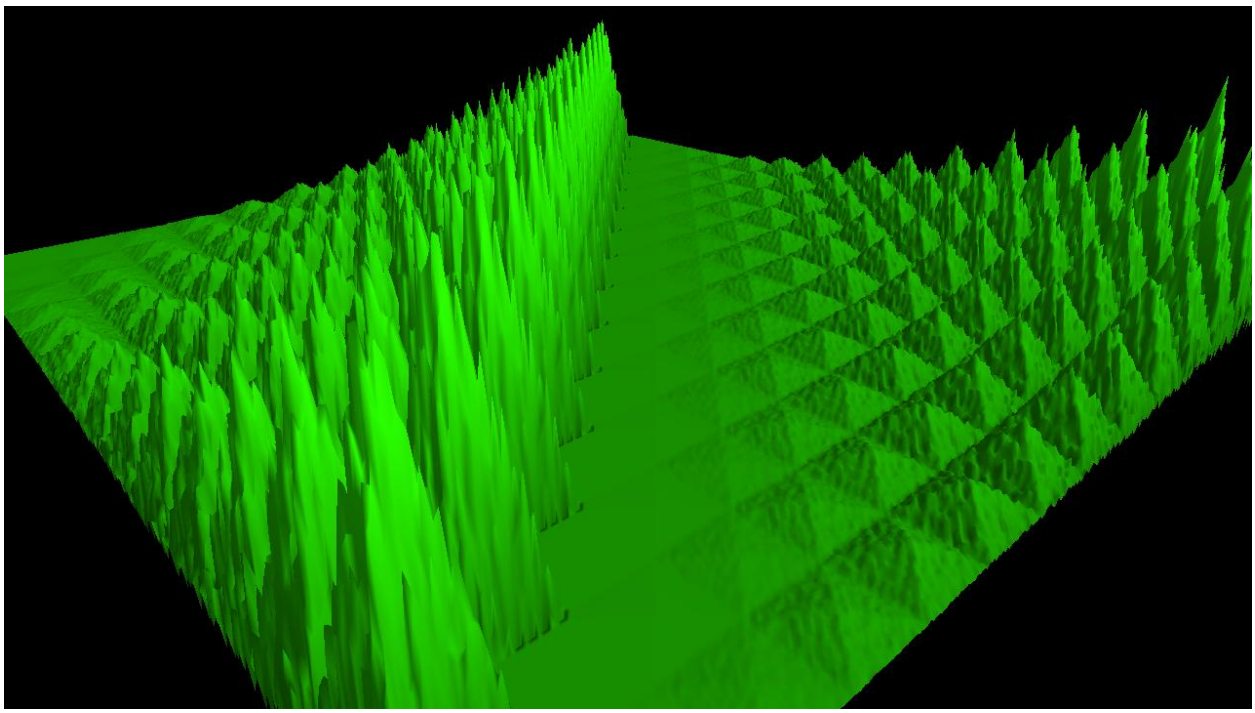

Figure 8 : Scène où sont réalisés les tests de performance

La complexité de la scène ne peut à elle seule générer toutes les mesures de performance. Une caméra automatisée vient complémenter la scène pour réaliser un échantillonnage diversifié. L'objectif est de programmer un parcours de caméra qui produira des images 3D avec des caractéristiques couvrant les 4 objectifs de test. Avant de poursuivre dans les détails de la trajectoire de la caméra, il est important de bien comprendre certains des concepts qui régissent une caméra dans un contexte d'image de synthèse :

1. Un modèle de caméra est toujours composé d'une coordonnée qui définit la position de l'observateur, d'une deuxième coordonnée qui indique le point visé par l'observateur et d'un vecteur spécifiant le haut de la caméra. Plus de détails techniques sur ces caractéristiques seront présentés dans la section 5.3.1.
2. Lorsque l'œil humain regarde une scène, les objets plus éloignés paraissent plus petits que les objets situés plus près, c'est ce qu'on appelle plus communément la notion de en perspective. Dans un contexte tridimensionnel, cet effet est obtenu via une projection perspective. Encore une fois, des détails techniques sur ce concept seront présentés dans la section 5.3.1.

La première étape pour définir une trajectoire de caméra automatisée est de spécifier le tronc de projection de celle-ci (plus de détails sur le tronc de projection sont disponibles à la section 5.3.1.2). Pour déterminer le volume de celui-ci pour notre banc de test, nous nous sommes basés sur l'application SVS que nous avons caractérisée. L'angle du champ de vision à la verticale (*Field Of View*, FOV) utilisé dans ce SVS est défini par un angle de 45 degrés et le format de l'image résultante suit un ratio 4:3. La distance du plan proche par rapport à l'observateur est de un (1) unité dans le repère cartésien du contexte OpenGL, soit la distance minimale. La distance du plan lointain par rapport à l'observateur est variable et dépend, quant à elle, des dimensions de la scène. En effet, lorsque nous exécutons notre banc de test, nous voulons être capables de générer des images qui incluent un maximum de sommets de la scène, tout en ayant le plus petit tronc de projection possible, afin de maximiser la précision du tampon de profondeur. Puisque le FOV de la caméra à une valeur fixe de 45 degrés, la hauteur significative maximale de la caméra est égale à la largeur de la scène. Passé cette limite et en faisant pointer la caméra vers le sol, on peut remarquer que la scène deviendra plus petite que la fenêtre d'affichage. La Figure 9 montre

bien cet effet non voulu. Dans un scénario d'utilisation typique d'un SVS, une telle situation n'arrivera jamais puisque « physiquement » un avion ne pourra pas se rendre une altitude assez haute pour voir la frontière de la scène. C'est donc pour cette raison que nous avons défini la hauteur maximale de la caméra comme étant égale à la largeur de la scène. Maintenant, il ne nous reste plus qu'à trouver le point le plus éloigné qui peut être visé à cette hauteur. La distance entre ce point et la position de la caméra à sa hauteur maximale et au milieu de la tuile centrale, deviendra la valeur du dernier paramètre du tronc de projection soit la distance du plan lointain. Lorsque la caméra est positionnée au milieu de la tuile centrale et à sa hauteur maximale, le point le plus éloigné qu'elle peut viser correspond à un des quatre coins de la scène (ce qui correspond à une demi-diagonale de scène sur le plan XZ). Sachant la hauteur maximale de la caméra et de la longueur de la demi-diagonale de la scène, on peut alors trouver la distance du plan lointain de la caméra grâce au théorème de Pythagore (voir Figure 10).

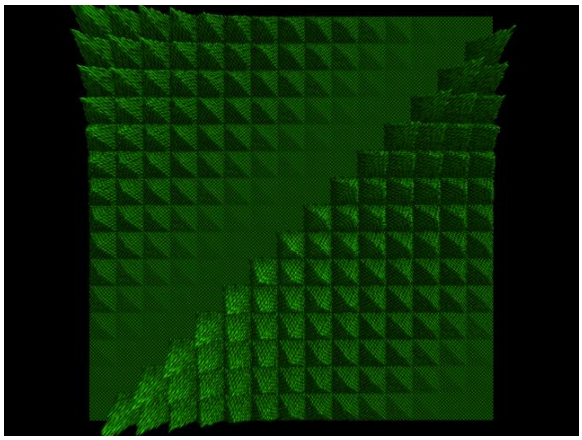

Figure 9 : Scène trop petite par rapport à la taille de la fenêtre

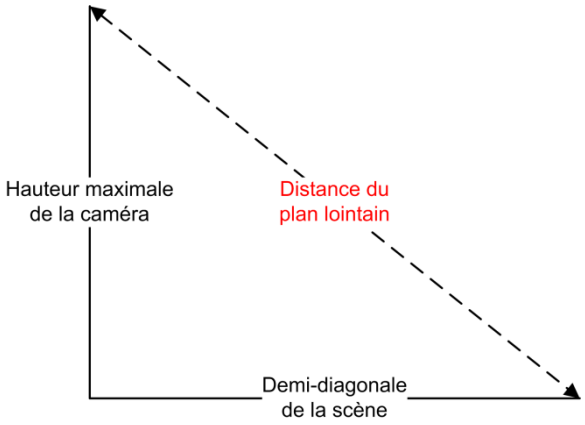

Figure 10 : Distance du plan lointain de la caméra

Avec le tronc de projection maintenant défini, nous pouvons procéder à la deuxième étape soit la définition de la trajectoire de la caméra. L'objectif ici est de générer des images 3D dans lesquelles on ajoute graduellement des sommets à afficher afin d'obtenir une gamme complète d'images qui couvre tous les cas d'utilisation réels. Nous discuterons de cette trajectoire à la section 5.3.3. D'autre part, les détails techniques des caractéristiques graphiques mentionnées

dans cette section seront abordés en profondeur dans la section 4.4. L'objectif ici était de bien visualiser les scénarios de test.

4.2.4 Gestion des résultats

Les tests réalisés par notre outil font varier chacune des propriétés présentées à la section 4.2.1 une à une afin de mesurer leurs impacts sur les performances de traitement graphique sur une plateforme cible. L'objectif ici est de représenter les résultats générés sous un format de sortie intuitif. Puisque tous ces paramètres affectent les performances du système, chaque test vise un seul paramètre à la fois afin de mesurer l'impact que chacun d'entre eux induit aux performances de traitement graphique. On regroupe les résultats produits selon le paramètre ciblé par un test. Pour chacun des paramètres testés, deux fichiers sont créés. Le premier fichier est une feuille de calcul de format csv qui regroupe toutes les données reliées au temps de traitement par image. Ce fichier sert à produire un diagramme illustrant les courbes de performance associées à chaque variation du paramètre ciblé par le test. Ces courbes représentent la variation du temps de traitement par image selon le pourcentage de la scène rendu à l'écran. Par conséquent, une courbe sera générée pour chaque variation du paramètre. Le deuxième fichier est un fichier texte dans lequel est inscrit un compte rendu du test effectué. Celui-ci contient des informations générales par rapport au type de test réalisé et d'autres informations plus précises par rapport aux variations du paramètre testé. Les informations générales sont un récapitulatif des paramètres fixes du test tandis qu'un ensemble de mesure est généré pour chaque variation du paramètre testé. Cet ensemble de mesure inclut les valeurs suivantes :

1. Temps de traitement par image moyen (exprimé en FPS);
2. Temps de traitement maximum pour une image (exprimé en IFPS);
3. Taille de la mémoire occupée par la scène (incluant les sommets, les index, les couleurs, les vecteurs normaux aux sommets et la texture);

Comme nous l'avons mentionné dans le dernier paragraphe, les temps de traitement par image sont mesurés selon le nombre de sommets rendus à l'écran. Nous avons procédé de la sorte

afin de visualiser la corrélation entre les performances de traitement graphique et la quantité d'information affichée à l'écran. Cependant, certains types de test génèrent des scènes avec des nombres de sommets variables. Parmi ces tests, nous retrouvons celui qui incrémente le nombre de sommets par tuile et celui qui incrémente le nombre de tuiles utilisées pour constituer la scène. D'autre part, le nombre de sommets n'est pas une unité simple d'interprétation et laisse place à beaucoup de subjectivité. Nous avons plutôt opté pour le pourcentage de la scène rendu à l'écran pour exprimer cette mesure. Pour obtenir cette unité, nous avons divisé le nombre de sommets inclus dans le tronc de projection par le nombre total de sommets qui constituent le terrain. Cette unité de mesure facilite la compréhension des résultats et fournit une échelle de mesure uniforme permettant de comparer les tests réalisés, peu importe le nombre de sommets qui constitue la scène.

Cette approche de classement des résultats a été retenue, car elle permet de comparer facilement les résultats générés par l'outil. Les diagrammes permettent de visualiser la dégradation des performances engendrée par la variation d'une propriété graphique puis le compte rendu indique les extrémums atteints lors de l'exécution des tests.

4.3 Descriptions des environnements de tests

Nous avons implémenté deux versions de notre outil, la première étant pour un environnement MS Windows et la deuxième pour un environnement Linux embarqué.

Le premier système sur lequel nous avons lancé nos tests est le prototype de plateforme avionique de notre partenaire industriel. Elle est munie du processeur Intel Core i7-2600 à 8 cœurs ayant une fréquence de 3,40GHz, de 4Go de mémoire vive (RAM) et est régie par le système d'exploitation Windows XP. La carte graphique (GPU) présente sur la plateforme est la Matrox M9148 LP PCIe x16. Cette dernière possède 4 ports de sortie vidéo pouvant atteindre une résolution maximale de 1920x1200 pixels chacun et 1024Mo de mémoire graphique pour le GPU. La librairie graphique supportée par ce système est OpenGL 2.0. Cependant, nous avons utilisé les librairies (SDK) du processeur PowerVR [33] développé par la firme Imagination Technologies pour nous permettre d'émuler la librairie OpenGL ES 1.4 sur la librairie graphique déjà présente sur la plateforme. L'émulation [34] de la librairie graphique nous permet d'utiliser

le même contexte d'exécution – ici, les librairies graphiques OpenGL ES – que nous avons développé dans un contexte embarquée sur une platforme commerciale. Notre partenaire a choisi ces composants matériels de pointe afin de minimiser l'impact du système d'exploitation (*Operating System*, OS) et du processeur générique sur les performances de traitement délivré par la carte graphique.

La deuxième plateforme que nous avons testée est un ordinateur à faible puissance de type « ordinateur sur carte » (*Single Board Computer*, SBC) nommé BeagleBoard-xM [35]. Celui-ci est doté d'un système sur puce (System on Chip, SoC) DM3730 [36] de la compagnie Texas Instruments qui lui-même intègre un microprocesseur ARM Cortex-A8 opérant à fréquence de 1GHz, 512Mo de RAM partagée avec le processeur graphique PowerVR SGX530 [37] de la compagnie Imagination Technologies. Le système d'exploitation opérant sur la plateforme est la distribution Angstrom [38] basée sur Linux™ et la librairie graphique supportée est la version 1.4 d'OpenGL ES. La version choisie de la librairie OpenGL devait refléter les considérations avioniques, la possibilité de pouvoir compiler sur un ordinateur commercial par compilation croisée, ainsi que l'existence de cette librairie pour l'environnement du système embarqué. La librairie graphique choisie est donc la version 1.4 d'OpenGL avec le profil système embarqués (ES). Le profil ES version 1.4 est relativement proche du profil système critique (SC), basé sur la version 1.3 d'OpenGL. Finalement, le SDK et l'environnement de compilation est compatible pour compiler sur Linux/Intel vers Linux/ARM. Cette version représente donc la solution optimale.

Le dernier système sur lequel nous avons exécuté notre outil est un ordinateur portable commercial : un Lenovo P580. Il est muni d'un processeur Intel Core i7-3520M à 4 cœurs ayant une fréquence d'opération de 2,90GHz, de 8Go de RAM et utilise Windows 7 comme système d'exploitation. L'ordinateur possède une carte graphique NVIDIA GeForce GT 630M. Celle-ci peut atteindre une résolution maximale de 1920x1200 pixels et possède 2048Mo de mémoire graphique. La librairie graphique présente sur le système est OpenGL 4.0 mais, encore une fois, nous avons utilisé l'émulateur de la librairie OpenGL ES 1.4 pour réaliser nos tests.

Un récapitulatif des principales caractéristiques des trois plateformes de test est présenté au Tableau 1.

Tableau 1 : Descriptions matérielles des plateformes de test

	Beagleboard	**Plateforme avionique**	**Ordinateur de bureau**
CPU	1 GHz ARM Cortex-A8	**Intel Core i7-2600 CPU @ 3,40GHz (8 cœurs)**	Intel Core i7-3520M CPU @ 2,90GHz (4 cœurs)
RAM	512Mo MDDR SDRAM (partagé avec le GPU)	**4Go DDR3**	8Go DDR3
GPU	Imagination Technologies PowerVR SGX530	**Matrox M9148 LP PCIe x16**	NVIDIA GeForce GT 630M
Mem. graphique dédiée	0	**1Go**	2Go
Bus mémoire	SDR 2,8 Go/s	PCI-e 16x 2.0 8 Go/s	PCI-e 16x 2.0 8 Go/s
OS	Linux Angstrom	**Windows 7**	Windows 7
Librairie graphique	OpenGL ES 1.4	**OpenGL 2.0 / OpenGL ES 1.4 émulée**	OpenGL 4.0 / OpenGL ES 1.4 émulée
Sortie vidéo	DVI	**DVI**	DVI

CHAPITRE 5
STRUCTURE ET FONCTIONNEMENT DU BANC DE TEST

Dans cette section, nous allons présenter le fonctionnement détaillé de notre banc de test. Pour en faciliter la conception, notre outil a été scindé en 5 blocs de traitement : un lecteur de tests, un générateur de scène, un générateur d'images, un analyseur d'images et une unité qui recueille les autres métriques de sortie. Le schéma ci-dessous illustre comment notre application fonctionne. Nous allons expliquer le rôle de chacune de ces étapes de traitement afin de bien comprendre comment le banc de test arrive à produire des résultats.

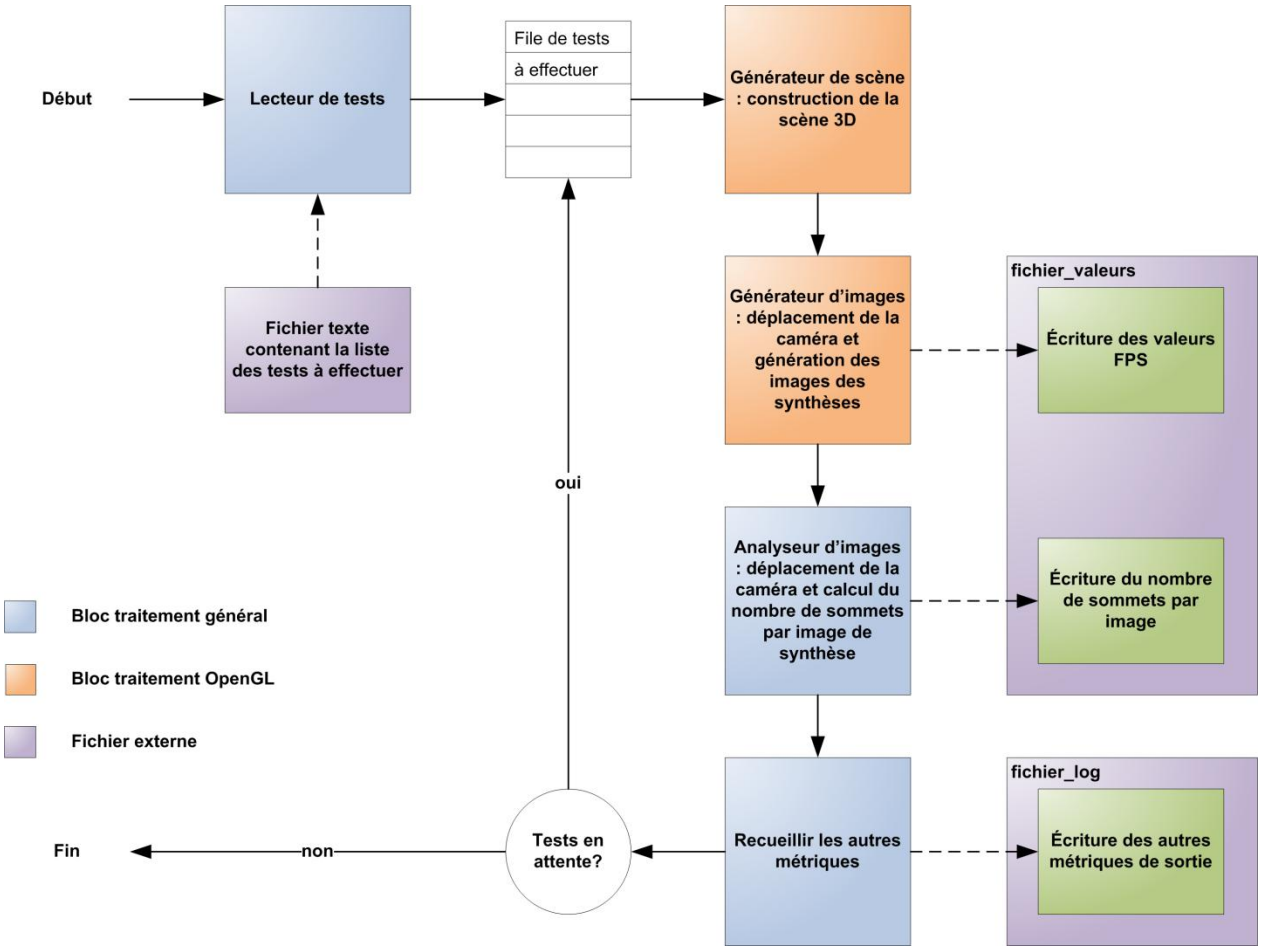

Figure 11 : Diagramme de blocs qui décrit le fonctionnement du banc de test

5.1 Lecteur de tests

Le lecteur de tests a pour utilité d'aller lire tous les tests inscrits dans le fichier de déclaration des tests nommé *tests-setup.txt*. C'est dans ce fichier, qui est situé dans le répertoire racine du programme, qu'un utilisateur entre tous les tests qu'il souhaite exécuter. Le lecteur de tests lira chacun de ces tests définis sous format texte et les transformera un à un en objet test (*STest*). Les objets *STest* sont alors stockés dans un tableau qui sera ultérieurement passé au prochain bloc de traitement qui les exécutera. De plus amples informations sur ce bloc fonctionnel sont disponibles à la section 0.

5.2 Générateur de scène

Le générateur de scène a pour mission de créer un contexte OpenGL dans lequel sera dessinée la scène de notre environnement virtuel. Cette opération consiste, en premier lieu, à créer une fenêtre qui accueillera ce contexte graphique et en deuxième lieu, à générer une scène tridimensionnelle dans laquelle nous pourrons effectuer des mesures de performance. Les paramètres servant à calibrer ce système viennent des objets *STest* en provenance du module lecteur de tests présenté dans la section précédente. Pour chacun des tests à exécuter, un nouveau contexte et une nouvelle scène sont créés (un test est réalisé par valeur). Nous allons maintenant expliquer comment la scène 3D est créée.

5.2.1 Génération d'une tuile de terrain

Comme nous l'avons mentionné plus tôt, notre scène est constituée d'un agencement de tuiles de terrain de forme pyramidale à base carrée. Chacune de ces tuiles est encapsulée dans un objet *CTile*. Ce dernier est défini par quatre paramètres :

1. La dimension d'un côté de tuile : Cette valeur *n* indique les dimensions de la base carrée de la pyramide seront de *n* par *n*. Elle est exprimée en unité de grandeur propre au repère cartésien d'OpenGL.
2. Le nombre de sommets par côté de tuile : Cette valeur *n* spécifie que la tuile sera composée de *n* par *n* sommets. Si on fait abstraction de l'effet visuel de perspective, la Figure 12 et la Figure 13 (prises de vue situées au-dessus et sur le côté de la pyramide)

montrent que les sommets qui constituent une pyramide sont à égale distance les uns des autres sur le plan XZ. L'effet de perspective éloigne les points situés près de l'observateur et rapproche ceux situés à une distance plus grande.

3. La hauteur de la pyramide : Cette valeur *n* indique que la taille de la pyramide sera de *n* unités OpenGL.
4. Le niveau de bruit à introduire sur la surface de la pyramide : Cette propriété vise à ajouter un effet d'ondulation aléatoire sur la surface de la tuile de terrain afin d'obtenir des cas de scénario plus réels et éviter que les pyramides aient des surfaces planes (voir Figure 14 et Figure 15). Cette valeur est encore une fois exprimée en unité cartésienne OpenGL et a un impact uniquement sur l'axe des Y. Selon la valeur *n* entrée, les positions verticales des sommets seront recalculées de sorte que leurs valeurs finales soient incluses dans la plage : position originale ± *n*.

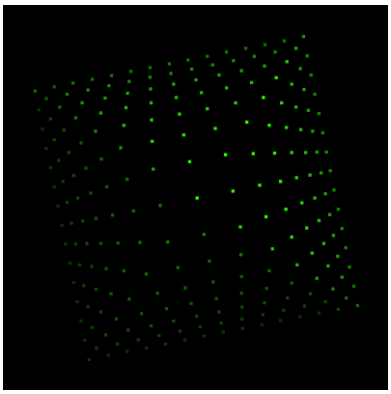

Figure 12 : Vue du dessus en pointillé

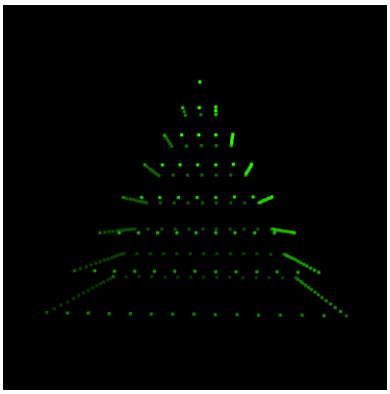

Figure 13 : Vue de côté en pointillé

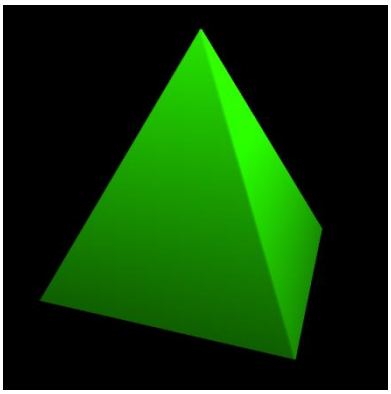

Figure 14 : Vue perspective

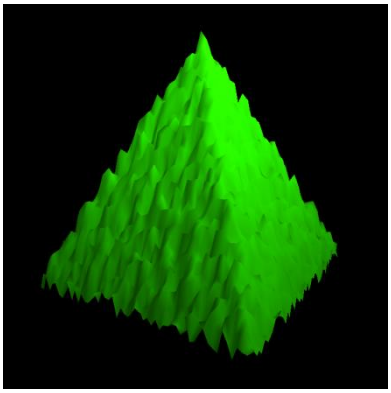

Figure 15 : Vue perspective avec bruit

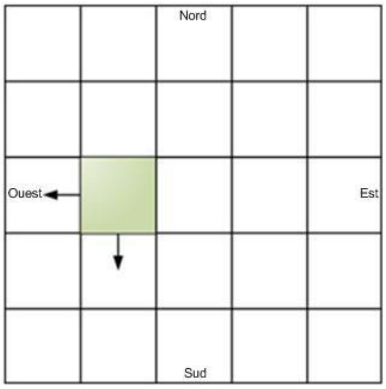

Figure 16 : Vue du dessus montrant les tuiles *ouest* et *sud* voisines d'une tuile de terrain

Comme nous l'avons mentionné plus tôt dans cet ouvrage, les coordonnées des sommets qui composent un objet de forme pyramidale à base carrée peuvent être facilement générées via une équation mathématique. Les points qui composent une tuile sont équidistants les uns des autres sur le plan XZ. On obtient la distance *nord-sud* et *est-ouest* entre chaque sommet avec l'équation suivante : (dimension d'un côté de tuile) / (nombre de sommets par côté de tuile). Puis, pour le calcul de leur altitude, nous nous sommes basés sur le code présenté en annexe à la section 0. Nous avons modifié la formule afin d'obtenir une surface irrégulière telle qu'illustrée à la Figure 15. Les seuls sommets qui échappent à l'application de cette formule sont ceux situés aux extrémités *ouest* et *sud* d'une tuile. Ceux-ci prennent plutôt la valeur Y du sommet adjacent de la

tuile voisine. Par exemple, les sommets limitrophes *ouest* et *sud* de la tuile en surbrillance à la Figure 16 prennent respectivement les valeurs Y des sommets limitrophes *est* de la tuile située à l'*ouest* et les valeurs Y des sommets limitrophes *nord* de la tuile située au *sud*. Il en est ainsi, car nous voulons que les tuiles forment une scène continue (sans disparité entre les tuiles). Dans le cas où une tuile n'aurait pas de voisin à l'*ouest* ou au *sud*, l'altitude des sommets est 0 (nulle) par défaut.

Une fois les sommets d'une tuile de terrain générés, il faut spécifier à OpenGL leur ordonnancement afin que l'engin graphique puisse générer des objets 3D. En effet, OpenGL SC ne peut générer que des points, des lignes ou des triangles pleins. Il faut donc lui fournir une table d'indexation des sommets pour que la librairie puisse dessiner les surfaces de la tuile de terrain. Le type de rendu qui nous intéresse ici est le triangle plein. Pour l'obtenir, il faut combiner 3 indices de sommet. Cette combinaison de trois sommets doit être faite de sorte à remplir complètement la surface d'une tuile de terrain de triangles. La Figure 17 présente l'agencement pour une tuile formée de 4 sommets par 4 sommets. Le premier triangle (dans le coin sud-ouest) est composé des sommets AEB, le deuxième des sommets BEF et ainsi de suite. Chacun des sommets est associé à un index qui varie de 0 à *n* selon l'ordre de définition établi à l'étape précédente. Pour former le carré AEFB, il faut donc envoyer à l'engin graphique l'ensemble d'indices {(0, 4, 1), (1, 4, 5)}. Par conséquent, pour dessiner une tuile de 4X4 sommets, il faudra envoyer une liste de 54 valeurs à OpenGL. Notons aussi que l'ordre dans lequel les sommets sont énumérés forme toujours une boucle dans le sens horaire. Cette subtilité permet de définir l'avant et l'arrière d'un triangle. Nous expliquerons à la section 5.3.2.2 comment tirer avantage de cette caractéristique.

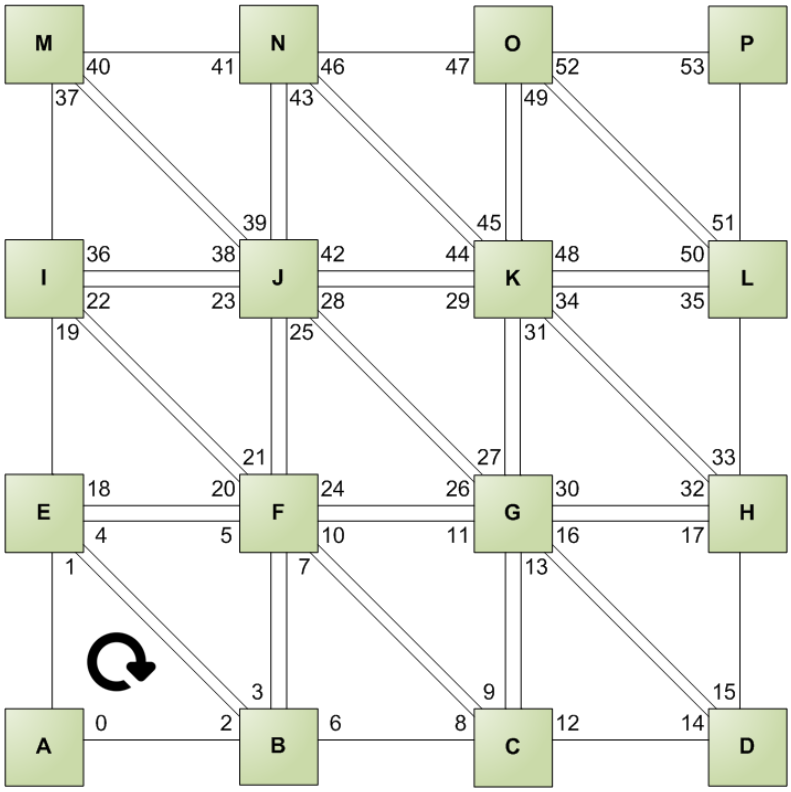

Figure 17 : Agencement des sommets pour une tuile de terrain composée de 4X4 sommets

Comme nous l'avons mentionné à la section 3.3, notre environnement virtuel utilise un modèle d'illumination de Gouraud. Ce modèle d'illumination permet d'effectuer un rendu de surface harmonieux et d'éviter le problème du passage brusque d'une couleur à une autre entre les polygones. Pour réaliser cet effet de lissage, l'illumination de chaque polygone est déterminée par interpolation linéaire des valeurs de l'intensité lumineuse aux sommets d'un polygone. Ces valeurs d'intensité sont basées sur les vecteurs normaux de chacun des sommets du maillage d'un objet 3D. Un vecteur normal est la somme normalisée de tous les vecteurs normaux à la surface des polygones composés de ce sommet. En sachant les vecteurs normaux des sommets d'un objet, nous pouvons obtenir l'intensité lumineuse à chacun de ces sommets puis, par interpolation linéaire, nous pouvons afficher des objets avec des formes courbes qui ont un dégradé homogène de couleur entre chacun des sommets évitant ainsi d'avoir des arrêtes franches entre chacun des polygones d'un objet. Les seules données à fournir à ce modèle d'illumination sont les vecteurs normaux aux sommets. La librairie graphique se charge des autres calculs à effectuer pour le

rendu final de l'image de synthèse. Les vecteurs normaux doivent être chargés en mémoire vidéo en même temps que le chargement de l'objet.

Contrairement au modèle d'illumination plat (normal) qui n'a qu'un seul vecteur normal par face, le modèle d'illumination de Gouraud se base sur les vecteurs normaux situés aux sommets des faces (triangles) pour interpoler des vecteurs normaux intermédiaires entre chacun des sommets. Comme le montre la Figure 18, un vecteur normal spécifie l'angle de réflexion d'une source lumineuse sur la surface d'un objet en un point donné. Lorsqu'on utilise un modèle d'illumination plat, OpenGL n'utilise qu'un seul vecteur normal par face. Ceci implique que chaque face aura une intensité lumineuse uniforme qui entrainera un effet de non-continuité sur les surfaces irrégulières (voir la sphère gauche à la Figure 19). Nous utilisons donc l'interpolation de Gouraud et ses vecteurs normaux intermédiaires pour briser cette uniformité lumineuse sur les faces (voir la sphère droite à la Figure 19).

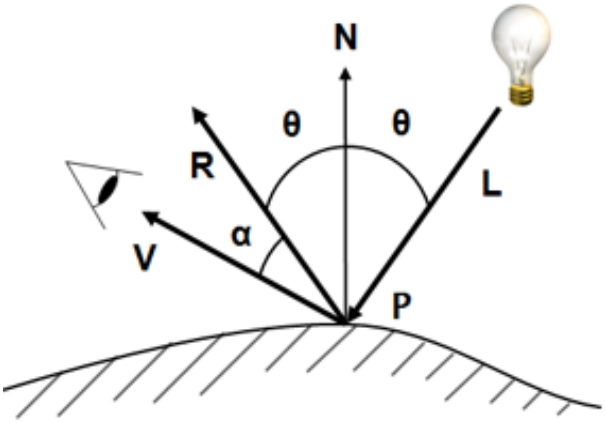

Figure 18 : Réflexion lumineuse par rapport à la normale d'une surface

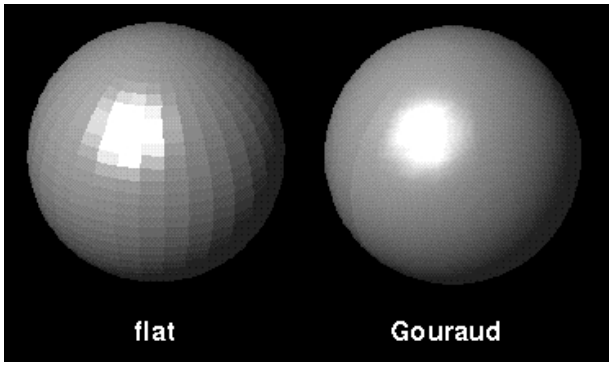

Figure 19 : Modèle d'illumination plat vs Gouraud

Si on fournit les vecteurs normaux aux sommets des faces (triangles) d'un objet 3D, l'engin graphique d'OpenGL interpolera les vecteurs intermédiaires pour réaliser le modèle de Gouraud. Expliquons maintenant comment nous calculons un vecteur normal à un sommet. La première étape consiste à calculer deux vecteurs coplanaires à une face adjacente au sommet ciblé. En nous référant à la Figure 20, supposons qu'une face est définie par des points *t1*, *t2* et *t3*. Imaginons ensuite les deux vecteurs coplanaires *v1* et *v2* définis respectivement par les points {*t1*, *t2*} et {*t1*,

t3}. Pour trouver le vecteur normal à cette surface, il suffit de calculer le produit croisé des vecteurs *v1* et *v2*. Par définition, il en résultera un vecteur perpendiculaire aux deux autres et donc normal au plan défini par les points *t1*, *t2* et *t3* (voir Figure 21). Le calcul pour trouver le vecteur normal *vn* est le suivant :

$$vn = v1 \times v2$$

$vn = [vnx, vny, vnz]$ où,

$$vnx = v1y \times v2z - v1z \times v2y$$

$$vny = v1z \times v2x - v1x \times v2z$$

$$vnz = v1x \times v2y - v1y \times v2x$$

Pour trouver le vecteur normal à un sommet, il faut répéter cette opération pour toutes les faces qui partagent ce sommet. En additionnant les vecteurs résultants, on trouve le vecteur normal au sommet. Cette opération est illustrée à la Figure 22. Le vecteur normal au sommet *v* a été obtenu en additionnant les vecteurs normaux *v12*, *v23*, *v34* et *v41*. L'exemple utilise des faces carrées, mais le principe fonctionne tout aussi bien avec des faces triangulaires (qui est notre cas). Finalement, la dernière étape consiste à normaliser le vecteur résultant, car OpenGL prend en considération la norme de ces vecteurs lorsqu'advient le calcul de l'éclairage de la scène.

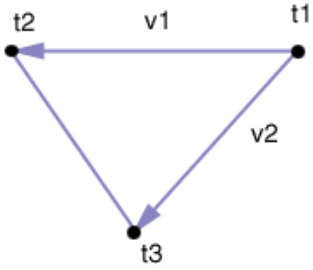

Figure 20 : Deux vecteurs coplanaires à une face

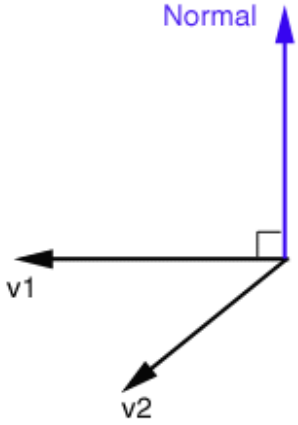

Figure 21 : Produit croisé de *v1* et *v2*

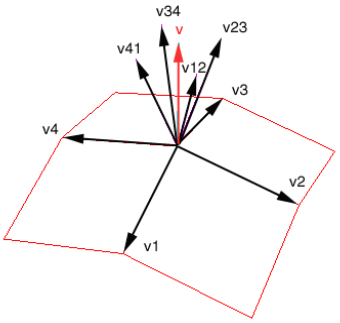

Figure 22 : Vecteur normal à un sommet

Toutes les coordonnées cartésiennes et vectorielles sont stockées dans des objets *CVec3f* soit une classe d'objet vecteur à 3 dimensions XYZ. Les valeurs qui définissent les 3 axes d'un vecteur sont de type *GLfloat* (un nombre à virgule flottante simple précision 32-bits). Toutes les coordonnées des sommets et tous les vecteurs normaux aux sommets sont stockés dans de tels objets. De plus, une couleur est associée à chacun des sommets. La valeur de cette couleur est traduite sous format RGB. Elles aussi sont stockées dans des objets *CVec3f*. Finalement, les indices des sommets sont stockés dans un tableau d'objets de type *GLshort* (des nombres entiers encodés sur 16-bits).

Nous avons évoqué à la section 3.2 qu'une tuile doit être en mesure de supporter deux niveaux de résolution : une haute et une basse. Pour afficher une tuile à haute résolution, il suffit de spécifier à OpenGL une table d'indexation des sommets qui prend en compte tous les sommets d'une tuile. Pour afficher une tuile à basse résolution, on effectue le même exercice, mais cette fois-ci en ignorant un sommet sur deux. Rappelons ici l'importance d'avoir un nombre de sommets par côté de tuile impair, car sinon il sera impossible de recouvrir complètement la surface de la tuile de triangle. En effet, la dernière rangée de sommets et la dernière colonne de sommets formeront une « crevasse » avec la prochaine tuile puisqu'il sera impossible de créer des triangles avec ces derniers. Pour avoir le choix d'afficher ces deux types de résolution dans l'environnement OpenGL, deux tables d'indexations doivent être chargées sur le GPU.

Le tableau suivant est un récapitulatif des données qui sont envoyées au GPU afin de dessiner une tuile de terrain. La taille en octet d'une tuile y est aussi spécifiée.

Tableau 2 : Données envoyées au GPU

Type de donnée	**Taille**	**Taille totale selon *n* sommets par côté de tuile**
Sommets	Trois valeurs à virgule flottante pour les 3 axes cartésiens. Il y en a un pour chacun des sommets. 3 X GLfloat 12 octets	$12 \times n^2$
Index des sommets (haute résolution)	Une valeur entière qui est l'indice d'un sommet. 1 X GLshort 2 octets	$6 \times 2 \times (n-1)^2$
Index des sommets (basse résolution)	Une valeur entière qui est l'indice d'un sommet. Un sommet sur deux est ignoré. 1 X GLshort 2 octets	$6 \times 2 \times \left(\frac{n-1}{2}\right)^2$
Couleurs des sommets	Une valeur RGB. Il y en a un pour chacun des sommets. 3 X GLfloat 12 octets	$12 \times n^2$
Vecteurs normaux aux sommets	Trois valeurs à virgule flottante pour les 3 axes cartésiens. Il y en a un pour chacun des sommets. 3 X GLfloat 12 octets	$12 \times n^2$
Taille totale d'une tuile de terrain en octets	Somme des 4 types de donnée.	$51n^2 - 30n + 15$

5.2.2 Génération de la scène (grille de tuiles)

L'objectif ici est de générer une grille de tuiles *CTile* (présentées à la section précédente) de sorte à former une scène qui respecte le scénario de test proposé à la section 4.2.3. Ces éléments seront à leur tour contenus dans un objet *CGrid* qui encapsule les tuiles qui définissent une scène. Une scène est définie par les quatre paramètres suivants :

1. Le nombre de tuiles par côté de scène : Cette valeur *n* spécifie que la scène sera composée de *n* par *n* tuiles de terrain et implique qu'une scène sera toujours de forme carrée.
2. La dimension d'un côté de tuile : Cette valeur est la même que celle présentée à la section précédente. Cependant, au lieu d'être affectée à une seule tuile, elle sera affectée à l'ensemble des tuiles qui composent la scène.
3. Le nombre de sommets par côté de tuile : Cette valeur est la même que celle présentée à la section précédente. Encore une fois, au lieu d'être affectée à une seule tuile, elle sera affectée à toutes les tuiles.
4. La profondeur des tuiles à haute résolution : Cette valeur *n* indique que toutes les tuiles autour de la tuile centrale (jusqu'à une distance de *n* tuiles) auront une haute résolution. Les tuiles subséquentes auront une basse résolution. Le diagramme présenté à la Figure 23 illustre bien cet étalonnage (les tuiles vertes étant à haute résolution et les tuiles grises à basse résolution).

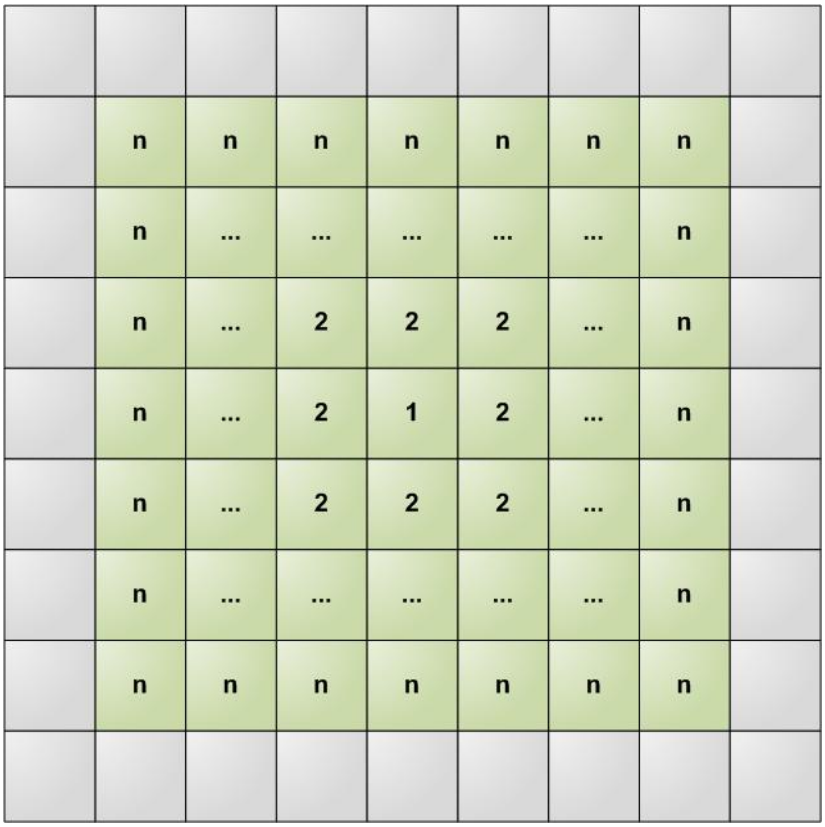

Figure 23 : Étalonnage des tuiles à haute et faible résolution

La première étape est de définir la hauteur maximale d'une pyramide. La plus haute a une altitude égale à 25% de la largeur de la scène. Ensuite, il faut étalonner les hauteurs de pyramide de façon exponentielle selon le schéma de la Figure 7 (présentée à la section 4.2.3). La tuile avec l'index le plus élevé aura la hauteur la plus élevée. Les autres hauteurs de pyramide sont basées sur la hauteur de pyramide la plus haute. Toutefois, pour chaque décrémentation d'index sur le schéma de la Figure 7, les pyramides verront leur hauteur être divisée par un facteur de 2. En prenant l'exemple de ce schéma et en supposant que *n* est l'indice d'une tuile, la hauteur d'une pyramide peut être obtenue avec l'équation suivante :

$$hauteur\ pyramide\ n = hauteur\ pyramide\ max \times \left(\frac{n}{index\ maximum}\right)^2$$

où l'index maximum serait la valeur 4 pour cet exemple

Afin que les pyramides gardent leur forme caractéristique, le bruit affecté aux surfaces de celles-ci est proportionnel à leurs hauteurs. En effet, chaque sommet qui compose une pyramide aura une valeur en Y aléatoire qui sera comprise entre ±10% de la hauteur de cette pyramide. Cette valeur est calculée à partir de la position « normale » en Y du sommet ciblé. Les figures Figure 24, Figure 25 et Figure 26 illustrent la progression du bruit selon la hauteur d'une pyramide. On remarque que le bruit proportionnel à la hauteur d'une pyramide permet de garder la forme caractéristique de ce type de prisme.

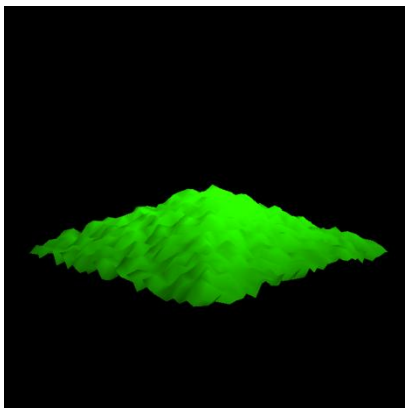

Figure 24 : Progression du bruit - bas

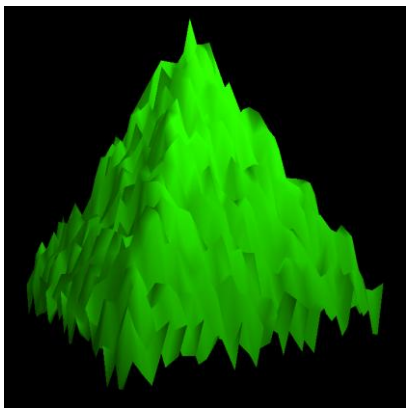

Figure 25 : Progression du bruit - moyen

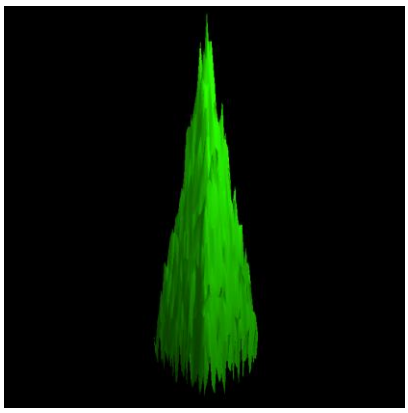

Figure 26 : Progression du bruit - haut

Le dernier élément à spécifier avant de charger la scène en mémoire RAM GPU est la texture qui recouvrira les tuiles. Tout comme l'application SVS étudiée, la scène de l'Outil n'utilisera qu'une seule texture. En effet, une seule texture est générée et celle-ci est ensuite appliquée sur chacune des tuiles de la scène. La première étape pour réaliser cette tâche consiste à générer la texture. La taille de celle-ci est exprimée en texel et elle provient de la valeur que l'utilisateur a entrée dans le fichier de déclaration des tests (*tests-setup.txt*). Un texel est le plus petit élément d'une texture appliquée à une surface (on peut le voir comme un pixel de texture). La librairie OpenGL SC impose l'utilisation de texture carrée avec des dimensions (en texels) qui sont des puissances de 2. Pour des raisons de simplicité, les textures que nous générons dans notre outil sont une alternance de texel de couleur noire et blanche (comme la mosaïque des cases d'un échiquier). Chacune des couleurs est encodée sous le format RGB 24-bits, soit 8-bits par couleur. La deuxième étape consiste à placer la texture sur une tuile. Cette opération est réalisée via une

table d'indexation qui contient les coordonnées en texels de la texture en chacun des sommets d'une tuile. Chaque sommet d'une tuile doit être associé à une coordonnée texel. Cette dernière est composée de deux nombres à virgule flottante simple précision 32-bits inclus entre les valeurs 0,0 et 1,0 (voir la Figure 27). Par exemple, le sommet situé à l'extrême sud-ouest d'une tuile serait associé à la coordonnée texel (0.0, 0.0), le sommet situé au milieu de la tuile à la coordonnée (0.5, 0.5), ou encore le sommet situé à l'extrême nord-ouest d'une tuile serait associé à la coordonnée (0.0, 1.0). Puisque les tuiles sont toujours composées du même nombre de sommets, on peut utiliser la même table d'indexation pour toutes les tuiles qui compose la scène.

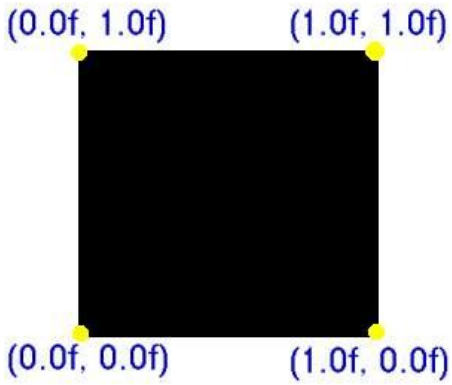

Figure 27 : Coordonnées texels d'une texture

Tableau 3 : Données de texture envoyées au GPU

Type de donnée	**Taille**	**Taille totale selon *n* sommets par côté de tuile et une texture de dimension m**
Texel	Valeur RGB composé de trois valeurs entières encodés sur 8-bits chacune. 3 X unsigned char 3 octets	$3 \times m^2$
Coordonnées de texture associée aux sommets	Valeur à deux dimensions qui spécifie une coordonnée de texture à un sommet d'un objet 3D. 2 X GLfloat	$8 \times n^2$

	8 octets	
Taille totale de la texture en octets	Somme des 2 types de donnée.	$3 \times m^2 + 8 \times n^2$

Avec tous ces éléments générés, nous avons maintenant une scène complète dans laquelle nous allons pouvoir effectuer nos tests de performance.

5.3 Générateur d'images de synthèse

Dans cette section, nous allons expliquer comment les images de synthèse sont générées. En premier lieu, nous allons définir la caméra qui nous permettra d'effectuer nos prises de vues. En second lieu, nous énumérerons la liste des différents réglages OpenGL qu'il faut spécifier afin d'obtenir un rendu 3D de qualité. Nous enchainerons ensuite avec la trajectoire automatisée de la caméra et préciserons comment les prises de vue sont obtenues. Enfin, nous terminerons en expliquant comment nous avons effectué nos mesures de temps de traitement par image.

5.3.1 Caméra

Afin de pouvoir contempler notre scène synthétique à l'écran, nous devons munir notre application d'un système qui nous servira de fenêtre vers le monde virtuel. L'objectif ici est de transformer les coordonnées des sommets des objets 3D qui composent la scène en coordonnées 2D interprétables par notre écran d'ordinateur (notre fenêtre vers ce monde). En graphisme, cette opération est appelée « transformation des sommets ». Le diagramme de blocs illustré à la Figure 28, montre les différentes étapes de transformation que doivent franchir les sommets lors de cette opération. Bien que la plupart de ces étapes sont prises en charge par OpenGL, il faut toutefois lui fournir la matrice caméra et la matrice de projection. Pour bien comprendre la provenance de ces matrices de transformation, nous expliquerons étape par étape les transformations de coordonnées mises en évidence dans le diagramme de blocs.

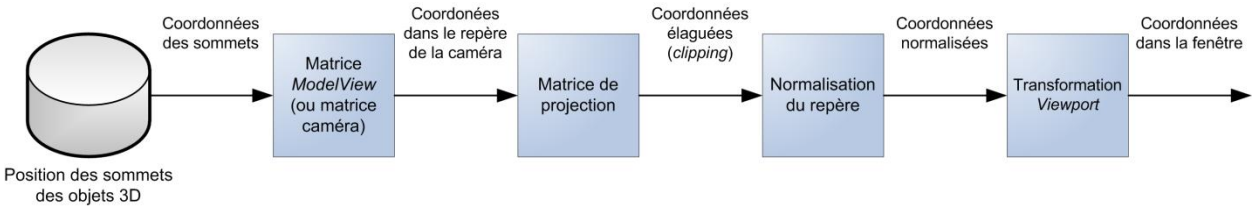

Figure 28 : Transformation des sommets dans OpenGL

5.3.1.1 Matrice ModelView

Les coordonnées des sommets représentent la position et l'orientation initiales des objets dans le repère de coordonnées locales d'OpenGL avant que toute transformation ne soit appliquée. OpenGL offre une série de fonctions permettant de modifier géométriquement un objet et ceci incluant la translation, la rotation ou encore le changement d'échelle (*scaling*). Ces transformations sont réalisées via la matrice (*Model*) gérée par OpenGL.

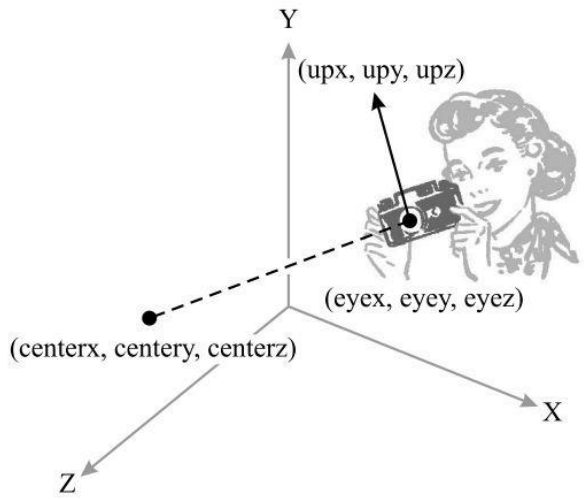

Figure 29 : Repère caméra

L'objectif maintenant est de redéfinir le repère de coordonnées locales en repère de coordonnées caméra. OpenGL se servira de ce repère pour savoir quoi dessiner à l'écran. Pour ce faire, nous devons lui fournir la matrice *View* qui servira à définir ce repère à chaque génération d'image. La Figure 29 illustre ce nouveau repère. On peut y remarquer l'ancien (celui composé

des trois axes XYZ) et le nouveau qui est défini par trois composantes : la position de l'observateur (*eye*), le point visé par l'observateur (*center*) et un vecteur spécifiant le haut de la caméra (*up*). Comme tout repère cartésien, le repère caméra est défini par trois axes et est orthonormé. Par convention OpenGL, le vecteur défini par les points *eye* et *center* deviendra le nouvel axe des Z négatif. Le vecteur défini par les points *eye* et *up* sera l'axe des Y positif et l'axe des X sera le vecteur orthogonal aux deux autres axes nouvellement définis. En normalisant les axes résultants, nous obtenons la matrice *View* suivante :

$$\begin{pmatrix} xcam_X & ycam_X & zcam_X & 0 \\ xcam_Y & ycam_Y & zcam_Y & 0 \\ xcam_Z & ycam_Z & zcam_Z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Où,

$$zcam = \frac{eye-center}{\|eye-center\|},$$

$$ycam = \frac{up-eye}{\|up-eye\|} \text{ et}$$

$$xcam = zcam \wedge ycam$$

Pour les tests que nous voulons effectuer, nous avons jugé les coordonnées cartésiennes peu efficaces. En effet, le parcours automatisé de caméra que nous avons introduit à la section 4.2.3 pour nos scénarios de tests fait souvent appel à des rotations de caméra ainsi qu'à des variations d'angle d'élévation. Chaque fois que nous voulons procéder à une rotation de la caméra, il en résulte un vrai casse-tête mathématique pour trouver les nouvelles coordonnées cartésiennes du point visé. Pour pallier à ce problème, nous avons plutôt opté pour l'utilisation de coordonnées sphériques. En effet, ce système a le net avantage d'orienter le repère caméra avec deux paramètres : l'angle de rotation θ et l'angle d'élévation δ (voir Figure 30). On peut ainsi entièrement contrôler la caméra avec seulement 3 paramètres : la position cartésienne de la caméra (XYZ), son angle de rotation (θ) puis son angle d'élévation (δ). Les formules mathématiques pour trouver les points *center* et *up* introduits dans le paragraphe précédent sont présentées ci-dessous. Elles sont exprimées en fonction de la position de la caméra *eye*, et les angles θ et δ. En multipliant la matrice *Model* par la matrice *View*, on obtient finalement la matrice *ModelView*.

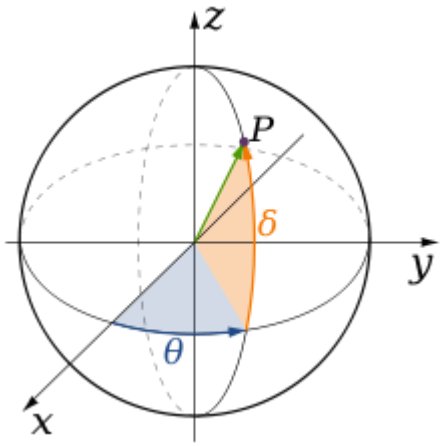

Figure 30 : Coordonnées sphériques

$$center_X = eye_X + \sin\left(\frac{\pi}{2} - \delta\right) * \cos\theta$$

$$center_Y = eye_Y + \cos\left(\frac{\pi}{2} - \delta\right)$$

$$center_Z = eye_Z + \cos\left(\frac{\pi}{2} - \delta\right) * \sin\theta$$

$$up_X = eye_X + \sin\left(\frac{\pi}{2} - \delta\right) * \cos\theta$$

$$up_Y = eye_Y + \cos\left(\frac{\pi}{2} - \delta\right)$$

$$up_Z = eye_Z + \cos\left(\frac{\pi}{2} - \delta\right) * \sin\theta$$

5.3.1.2 Matrice de projection

Un écran d'ordinateur est une surface 2D. Il est donc impossible d'y représenter l'ensemble d'une scène 3D qui, techniquement, n'a pas de frontière. On définit plutôt un volume que l'on souhaite projeter à l'écran nommé volume de projection. Tous les objets 3D situés à l'intérieur de ce volume seront dessinés à l'écran, tandis que les autres seront éliminés via une opération d'élagage (*clipping*). Puisque nous souhaitons générer des images avec des effets de perspective, nous spécifions un volume de forme pyramidale au sommet tronqué. Le volume d'observation est défini par les 4 paramètres suivants :

1. L'angle du champ de vision à la verticale (*field of view*, FOV);
2. Le format de l'image ou ratio de l'image $\left(\frac{largeur}{hauteur}\right)$;
3. La distance du plan proche par rapport à l'observateur;
4. La distance du plan lointain par rapport à l'observateur.

La Figure 31 représente un tronc de projection perspective. Notons que la hauteur des plans est obtenue explicitement via le FOV de la caméra et la distance des plans par rapport à la position de l'observateur. La largeur des plans est, quant à elle, obtenue avec le format de l'image

spécifié (un ratio 4:3 dans notre cas). En sachant ce ratio et la hauteur du plan, il est possible de déduire la largeur de celui-ci avec une simple règle de trois.

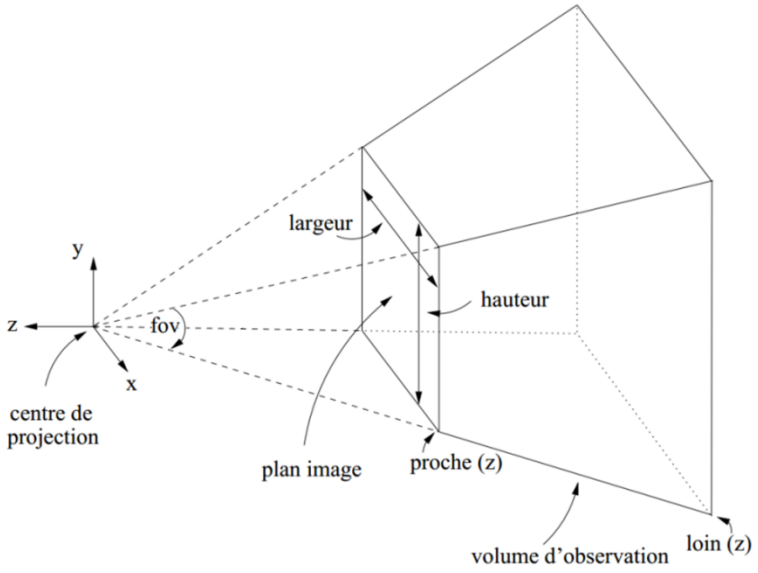

Figure 31 : Volume d'observation aussi appelé tronc de projection

La matrice de projection qui découle de ce volume est celle ci-dessous. La démonstration de celle-ci n'est pas faite dans cet ouvrage. Notons toutefois qu'elle est disponible dans la plupart des manuels qui traitent d'OpenGL. Donc, en prenant en considération le volume présenté précédemment, voici la matrice de transformation à fournir à OpenGL pour obtenir une projection perspective :

$$\begin{pmatrix} \frac{2 \times proche}{largeur} & 0 & 0 & 0 \\ 0 & \frac{2 \times proche}{hauteur} & 0 & 0 \\ 0 & 0 & \frac{proche + loin}{proche - loin} & -1 \\ 0 & 0 & \frac{2 \times proche \times loin}{proche - loin} & 0 \end{pmatrix}$$

5.3.1.3 Normalisation du repère

Suite à la définition du tronc de projection et à l'élagage des objets 3D qui se trouvent hors de ce volume, nous nous retrouvons maintenant avec la sous-section de la scène qui est définie à l'intérieur de ce tronc. Pour réaliser l'effet de perspective, OpenGL va ensuite normaliser ce volume (*Normalized Device Coordinates*, NDC). La Figure 32 illustre cette transformation du repère « caméra » vers le repère « normalisé ». Lors de cette opération, les coordonnées des sommets des objets 3D seront redéfinies dans un repère cartésien ayant les axes des X, Y et Z définis entre les valeurs 1 et -1.

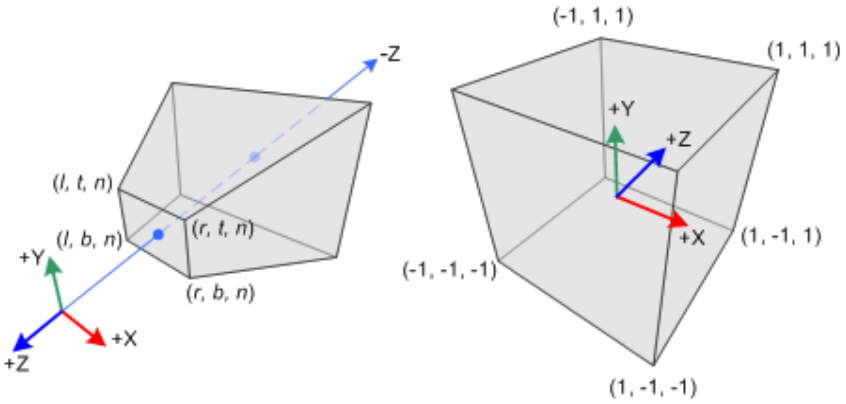

Figure 32 : Transformation de la projection perspective vers un repère normalisé (*Normalized Device Coordinates*, NDC)

Grâce à cette déformation du tronc de projection, les objets situés près de la caméra (du point de vue de l'observateur) paraitront plus gros et ceux qui sont plus éloignés paraitront plus petits. Les figures Figure 33 et Figure 34 illustrent cette déformation. C'est de cette façon qu'est réalisé l'effet de perspective lors de la génération d'images de synthèse.

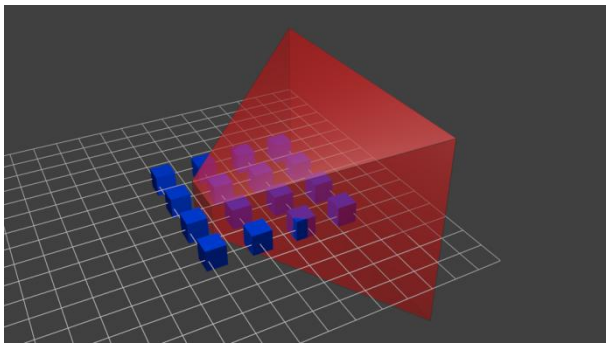

Figure 33 : Volume de projection avant la normalisation

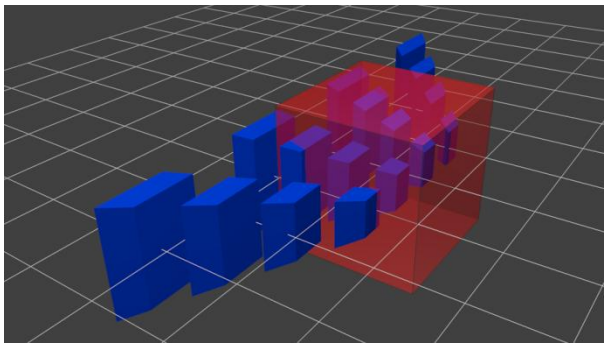

Figure 34 : Volume de projection après la normalisation

5.3.1.4 Transformation Viewport

Cette dernière étape consiste à mettre à l'échelle le repère NDC avec les dimensions de la fenêtre de projection. Cette étape est elle aussi prise en charge par OpenGL. Suite à cette dernière transformation, les sommets des objets 3D, maintenant exprimés en coordonnées fenêtre, seront acheminés au processus de rastérisation du pipeline graphique d'OpenGL afin de calculer ultimement la couleur finale des pixels projetés sur la fenêtre de rendu.

Les méthodes servant à créer et piloter une caméra ont été encapsulées dans une classe nommée *CCamera*. Il existe une seule version de cette classe puisque sa définition est entièrement indépendante du système d'exploitation sur lequel le banc de test est exécuté.

5.3.2 Réglages OpenGL

Pour bénéficier d'images de synthèse de bonne qualité et optimisées, il faut spécifier certaines propriétés à OpenGL. Encore une fois, ces réglages proviennent de l'analyse que nous avons effectuée sur l'application présentée à la section Chapitre 3. Ces derniers seront présentés dans cette section.

5.3.2.1 Tampon de profondeur

Lorsque l'on commande à OpenGL de dessiner un objet 3D, celui-ci va procéder selon l'ordre dans lequel les faces (triangles) de l'objet ont été définies. Le problème avec cette méthode de rendu est quelle ignore la profondeur (dans le repère cartésien) des faces. Donc en opérant de la sorte, il se pourrait fort bien qu'une face située à l'arrière-plan soit au final dessinée en avant-plan. Ce phénomène est bien illustré sur les figures suivantes. À la Figure 35, on voit que les deux premières faces d'un cube sont dessinées (verte et rouge). À la Figure 36 on vient dessiner une troisième face du cube (en bleu). Puisque cette face a été définie en troisième position, on obtient cette anomalie visuelle. Pour éliminer ce problème, il faut spécifier à OpenGL un tampon de profondeur.

Le tampon de profondeur stocke la profondeur de chaque pixel affiché à l'écran. Pour faire le parallèle avec un repère cartésien, la position XY représente la position du pixel dans la fenêtre de projection puis la valeur Z représente la profondeur de l'élément dessiné dans ce pixel (d'où sa nomenclature anglaise *z-buffer*). Chaque fois qu'OpenGL demande de dessiner un pixel à un endroit, il compare la profondeur du point à afficher et celle présente dans le tampon. Si le nouveau pixel est situé devant l'ancien, alors celui-ci est dessiné et la valeur de la profondeur dans le tampon est mise à jour. Dans le cas contraire, cela signifie que le pixel est derrière l'ancien et qu'il n'a pas lieu d'être affiché. Les valeurs du tampon de profondeur sont des valeurs encodées sur 16-bits et chacune d'entre elles représente une distance située entre les plans proche et lointain du tronc de projection (voir section 5.3.1.2). C'est pourquoi il est important de définir un tronc de projection avec une profondeur adaptée aux dimensions de la scène à rendre à l'écran afin de maximiser la précision de celui-ci.

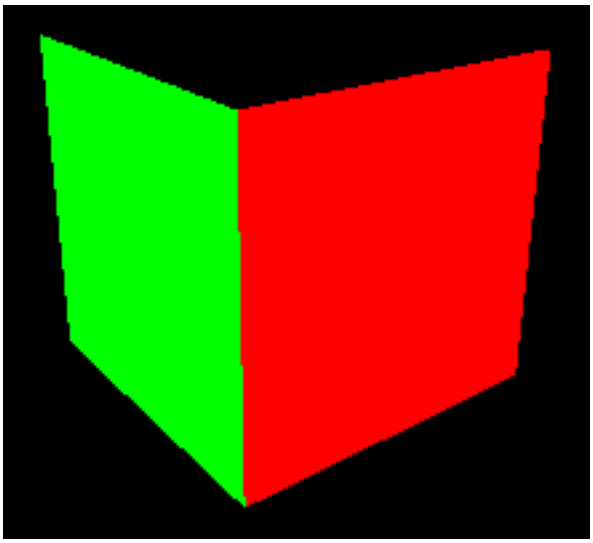

Figure 35 : Rendu d'un cube 1

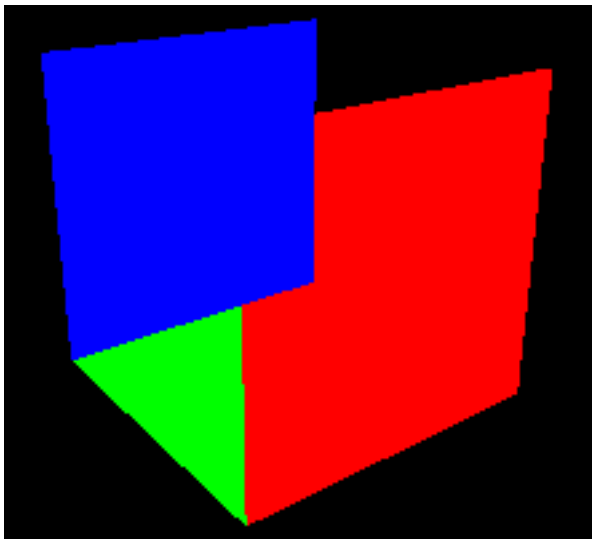

Figure 36 : Rendu d'un cube 2

5.3.2.2 Opération de culling

Le *culling* consiste à dessiner uniquement les faces (triangles) dont l'ordre des points est défini dans le sens des aiguilles d'une montre ou dans l'autre sens. L'intérêt de cette opération est l'optimisation. L'objectif ici est de toujours dessiner les faces extérieures d'un objet et de filtrer les faces intérieures (qui sont cachées) avec le *culling*. Lorsqu'OpenGL dessine un objet, les faces sont toujours affichées dans le même ordre, peu importe si elles sont visibles ou non. C'est le test de profondeur (mentionné dans la section précédente) qui détermine quelles faces seront affichées lors du rendu final de l'image. Cependant, ce test est très gourmand en puissance de calculs graphiques. Il est donc judicieux d'éliminer préalablement ces faces non visibles via l'opération de *culling* avant de commencer le rendu de la scène. De cette façon, OpenGL n'effectuera aucun traitement sur ces faces et ceci libérera par conséquent de la puissance de calcul. C'est donc pour ces raisons que nous avons porté une attention particulière à l'ordre d'agencement des sommets lors de la définition des tables d'indexations des sommets des tuiles (voir Figure 17 et la section 5.2.1).

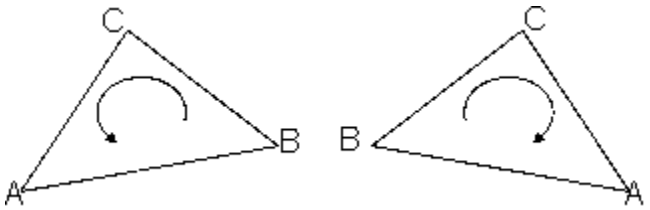

Figure 37 : Sens des sommets qui définissant une face (triangle)

5.3.2.3 Anticrénelage

Les effets de crénelage apparaissent dans une image généralement aux bordures d'un élément graphique. Ces bordures semblent alors morcelées ou déchiquetées. Ce phénomène est souvent appelé « effet escalier » vu la forte ressemblance de ces artéfacts avec la forme d'un escalier (voir dessin de gauche à la Figure 38). Pour éliminer cette distorsion visuelle désagréable, il faut munir l'application graphique d'un système d'anticrénelage. Une technique bien connue pour réaliser cette tâche en OpenGL ES est la méthode nommée anticrénelage par multiéchantillonnage (*Multisampled Antialiasing*). Cette technique consiste à diviser chaque pixel en un ensemble d'échantillons qui seront traités comme des « minis pixels » lors de l'étape de rastérisation. La couleur finale d'un pixel sera obtenue ultérieurement en calculant la couleur moyenne de ces échantillons. Notons que cette méthode n'est pas disponible dans la version critique de la librairie graphique OpenGL SC. Aucune méthode d'anticrénelage n'est actuellement définie pour cette version de la librairie graphique. Cependant, selon [14], la fonctionnalité d'anticrénelage par multiéchantillonnage sera ajouté dans la prochaine version d'OpenGL SC, ce qui explique pourquoi nous avons jugé pertinent de l'ajouter dès maintenant à l'Outil.

Figure 38 : lignes avec crénelage (à gauche) et lignes avec anticrénelage (à droite)

5.3.2.4 Effet de brouillard

L'objectif de cet effet est de flouter les frontières de la scène afin d'éviter que l'utilisateur ne perçoive la fin abrupte du terrain (voir les figures Figure 39 et Figure 40). Le brouillard utilisé dans notre application est de type linéaire c'est-à-dire qu'il commence à une profondeur A et s'obscurcit graduellement jusqu'à une profondeur B. Passée la profondeur B, le brouillard est complètement opaque. Notons que nous parlons ici de profondeur. Il en est ainsi, car les effets de brouillard sont calculés à partir du point de vue de l'utilisateur (position de la caméra). Dans ce repère, les axes X et Y sont la surface de la fenêtre et l'axe des Z est celui qui « rentre » dans cette fenêtre. Pour générer un effet de brouillard, on définit donc deux profondeurs sur cet axe des Z : une de départ et une de fin.

La distance qui sépare la caméra par rapport aux frontières de la scène dépend directement de la position sur la scène de celle-ci. Par exemple, plus l'altitude de la caméra est élevée, plus la profondeur du brouillard devra être éloignée. Si ce principe n'est pas respecté, passée une certaine altitude de caméra, la scène finira éventuellement par disparaitre dans le brouillard (voir Figure 41). On parle donc d'un effet brouillard dynamique. Nous avons pris en compte cette considération dans l'implémentation de notre outil. La profondeur du brouillard est calculée avant chaque génération d'image afin de flouter uniquement les frontières de la scène.

Tous comme anticrénelage par multiéchantillonnage, l'effet de brouillard n'est actuellement pas disponible dans la version critique OpenGL SC. Nous l'avons ajouté à la version actuelle de l'Outil, car encore une fois selon [14], la fonctionnalité de brouillard sera ajouté dans la prochaine version d'OpenGL SC.

Il est important de noter ici qu'un API n'est pas une finalité en soit. Si une fonctionnalité requise pour le développement d'une application graphique est manquante dans un API graphique (comme l'application SVS de notre partenaire par exemple), il est possible de simplement l'ajouter. Cela limitera toutefois le port de l'application vers d'autres plateformes supportant la librairie graphique OpenGL SC, car il faudra modifier les librairies graphiques sur ces nouvelles plateformes de sorte qu'elles puissent supporter les fonctionnalités de brouillard et d'anticrénelage.

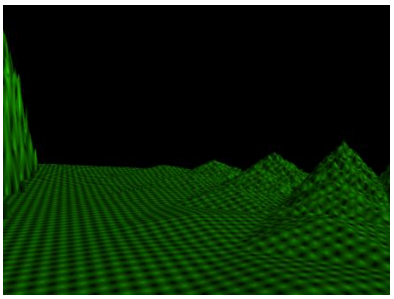

Figure 39 : Frontière d'une scène sans brouillard

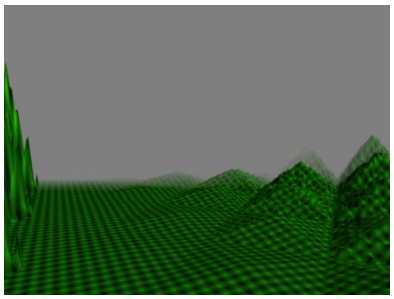

Figure 40 : Frontière d'une scène avec brouillard

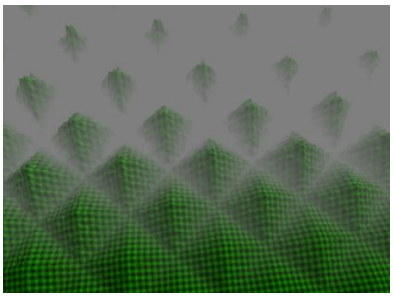

Figure 41 : Effet brouillard avec une profondeur fixe

5.3.3 Déplacement de la caméra et prises de vue

Pour générer nos images, nous avons muni notre banc de test d'une caméra à trajectoire automatisée qui ira prendre de prises de vues de la scène à des endroits stratégiques. Comme nous l'avons mentionné plus tôt dans ce document, l'objectif ici est de générer des images 3D dans lesquelles on ajoute graduellement des sommets à afficher afin d'obtenir une gamme complète d'images qui couvre tous les cas d'utilisation réels.

La première étape pour définir la trajectoire de la caméra est de déterminer l'espace dans lequel elle évoluera. Suite à la caractérisation que nous avons menée sur un prototype d'SVS à la section Chapitre 3, nous avons défini que la caméra serait confinée sur la tuile centrale sur plan horizontal XZ. De plus, nous avons convenu à la section 4.2.3 que la hauteur maximale de la caméra serait égale à la largeur de la scène. Cependant, l'utilisateur est libre de définir une hauteur de départ moins élevée que la hauteur maximale permise. La deuxième étape était d'imaginer une chorégraphie de caméra qui allait permettre de diversifier au maximum la quantité d'informations contenues dans les images. L'approche retenue est la suivante :

1. On positionne la caméra au milieu de la tuile centrale à la hauteur de départ définie par l'utilisateur.
2. On fait pivoter la caméra sur elle-même tout en faisant varier son angle d'inclinaison. Pour réaliser nos tests, nous avons opté pour une résolution de 32 images par tour. Celle-ci nous fournissait un ensemble d'échantillons adéquat pour l'analyse des performances.

Notons qu'il est possible de changer facilement cette résolution dans le code source de l'application.

3. Comme nous l'avons mentionné, lors de la rotation de la caméra sur elle-même, nous faisons aussi varier l'angle d'inclinaison de cette dernière. Cette variation n'est pas aléatoire. Elle dépend plutôt de la hauteur de la caméra et de la résolution d'images par tour. Pour la génération de la première image, on fait pointer la caméra vers la bordure de la scène (voir les flèches à la Figure 42). Ensuite, pour les images générées subséquemment, on diminue graduellement l'angle d'inclinaison de la caméra jusqu'à ce qu'elle pointe directement vers le sol (voir le point vert sur cette même figure).
4. On répète ensuite cet exercice pour plusieurs hauteurs à intervalle régulier. Nous avons choisi 8 hauteurs différentes pour réaliser nos tests et une hauteur de départ égale à 70% de la hauteur maximale de la caméra. Il est possible de modifier rapidement le nombre de hauteurs via le code source et la hauteur de départ via le fichier de déclaration des tests.

Nous avons opté pour cette approche suite à plusieurs essais. Celle-ci nous offrait un échantillonnage riche couvrant tous les cas de complexité du rendu graphique. La Figure 42 nous montre un exemple de trajectoire de caméra à 4 paliers. La hauteur de départ est plus petite que la hauteur de caméra maximale (point rouge). À chacun des 4 paliers, on remarque que l'angle d'inclinaison initial de la caméra pointe toujours vers la bordure de la scène. Entre chaque génération d'image (et par palier), le programme décrémentera graduellement cet angle d'inclinaison jusqu'à ce que la caméra pointe vers le sol (point vert).

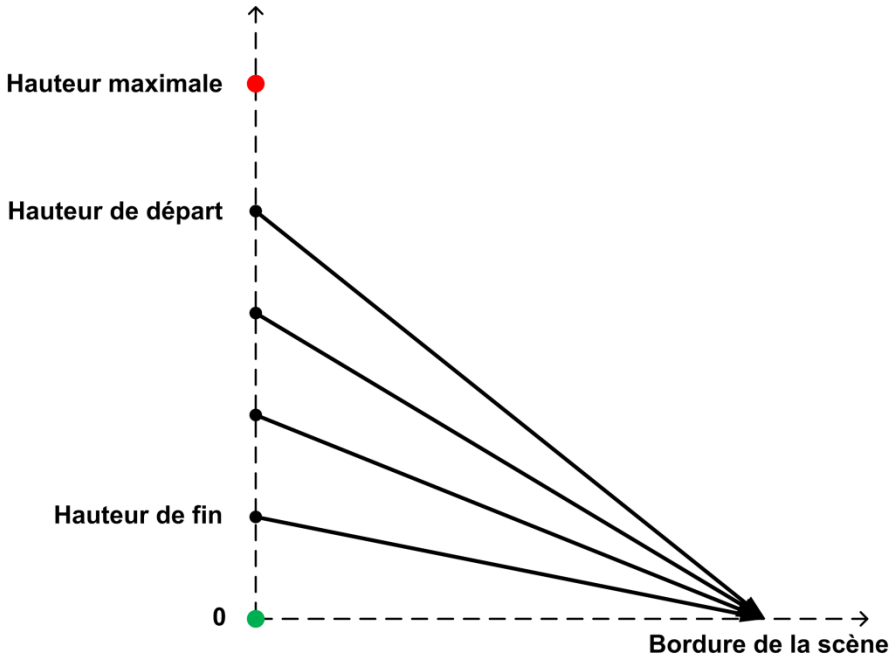

Figure 42 : Trajectoire de la caméra

5.3.4 Calcul du temps de traitement par image

Pour les raisons énumérées à la section 4.2.2, nous utilisons dans notre banc de test le temps de traitement par image comme mesure de performance. Selon le groupe Khronos [39] qui chapeaute les spécifications des API OpenGL, certaines notions importantes [40] sont à prendre en considération lors de ce calcul. En effet, une des erreurs les plus communes est de mesurer uniquement le temps pris pour dessiner l'image. Or, les implémentations d'OpenGL sont la plupart du temps pipelinées ce qui signifie que lorsqu'on commande à OpenGL de dessiner un élément, il n'est pas exact d'assumer que le rendu de l'image sera terminé lors du retour de la fonction. Typiquement, lorsqu'on commande à OpenGL de dessiner des objets, celui-ci va stocker dans une queue située à l'entrée de la carte graphique l'information à dessiner. Ensuite lorsque le processeur graphique est libéré, celui-ci prend les données contenues dans cette queue puis les dessine à leur tour. Le temps qu'OpenGL prend pour déplacer les données vers cette queue nous en dit peu sur les performances graphiques de l'application. On nous propose donc d'utiliser la fonction *glFinish* pour éviter ce calcul imprécis. Cette dernière est une fonction bloquante qui dépend de l'état graphique d'OpenGL. En effet, le retour de cette fonction

s'effectue uniquement lorsque toutes les opérations de rendu sont terminées. On utilise cette fonction avant de partir le minuteur et avant de l'arrêter et de cette façon, nous pouvons calculer avec précision le temps pris pour effectuer le rendu d'une image. Afin de minimiser l'intrusion de cette sonde logicielle, les opérations contenues dans ce laps de temps sont le rendu de l'image puis le déplacement de la caméra. Les opérations d'écritures des résultats sont effectuées une fois la minuterie arrêtée.

5.4 Analyseur d'images

Ce module a pour objectif de calculer le nombre de sommets présents dans les images générées par le module précédent. Cette étape est effectuée après la génération d'images afin de ne pas induire d'erreur dans le calcul du temps de traitement par image. Lors de l'accomplissement de cette opération, il n'est pas nécessaire de générer les images. On garde plutôt les objets qui constituent la scène en mémoire puis on refait le parcours de caméra uniquement afin de calculer le nombre de sommets présent dans le tronc de projection de celle-ci.

À chaque déplacement de la caméra, on vérifie la position de tous les sommets de la scène. Chaque fois qu'un sommet se trouve dans le tronc de projection de la caméra, on incrémente un compteur. La récolte de ces données nous permettra de générer des graphiques de performances qui nous donneront le temps de traitement par image selon le nombre de sommets traités lors de génération de cette image.

Pour identifier si un sommet est présent dans le tronc de projection, il faut s'assurer que le sommet en question est compris à l'intérieur du volume défini par les 6 plans du tronc de projection (voir Figure 43). Pour faire cette vérification, nous avons utilisé les propriétés du produit mixte [41] de trois vecteurs. Cette opération mathématique consiste à faire le produit scalaire d'un vecteur avec le produit croisé des deux autres. En géométrie, cette opération peut être interprétée comme le volume signé d'un parallélépipède. La Figure 44 montre un exemple de cette forme et est définie par les vecteurs a, b et c. L'ordre dans lequel s'effectuent les opérations de produit scalaire et de produit croisé est non permutable. Pour trouver le volume du parallélépipède décrit à la Figure 44, il faut faire le produit scalaire du vecteur a avec le produit vectoriel des vecteurs c et b : $a \cdot (c \times b)$. Tel que nous l'avons mentionné plus tôt, le résultat est

le volume signé de la forme. En effet, si on reprend le même exemple, mais qu'on y inverse les vecteurs *c* et *b*, cela aura pour effet de rendre le résultat de l'opération négatif. D'autre part, si le vecteur *a* avait pointé vers le bas plutôt que vers le haut, cela aussi aurait engendré un résultat négatif.

Pour trouver si un point est contenu à l'intérieur du volume du tronc de projection (Figure 43), il suffit de faire le rapprochement d'un plan du tronc avec le plan défini par les vecteurs *c* et *b* du parallélépipède puis la position du point par rapport à un plan du tronc avec le vecteur *a* du parallélépipède. Pour savoir si un sommet est inclus dans le tronc de projection, il faut faire pour chacun des 6 plans qui définissent le tronc, le produit mixte de deux vecteurs qui définissent le plan avec le vecteur qui définit la position du sommet à vérifier. Les trois vecteurs doivent être calculés avec le même point de référence (voir le point initial des trois vecteurs à la Figure 44). Ensuite, il s'agit d'orienter le « haut » des plans nouvellement définis, tous vers l'intérieur du tronc de projection. Finalement, pour vérifier que le sommet est bel et bien situé à l'intérieur du tronc, il suffira de vérifier que les résultats des 6 produits mixtes soient tous positifs.

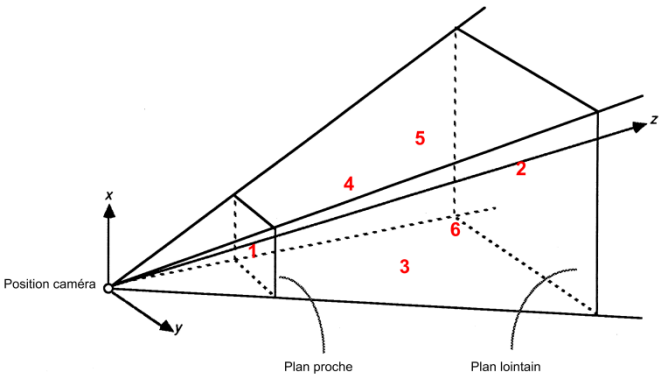

Figure 43 : Les 6 plans du tronc de projection

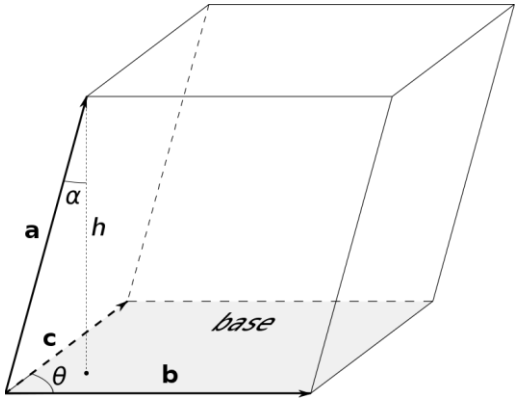

Figure 44 : Parallélépipède

CHAPITRE 6
RÉSULTATS ET DISCUSSION

Les expérimentations que nous avons réalisées ont pour objectif principal de nous permettre de visualiser rapidement l'impact sur les performances graphiques d'un système graphique avionique en faisant varier les différents paramètres présentés à la section 4.2.1. Dans cette section, nous allons présenter en premier lieu les différents résultats que nous avons obtenus avec l'Outil (Figure 11 du Chapitre 5). Nous poursuivrons avec l'analyse et une discussion générale portant sur les performances graphiques d'un système.

6.1 Résultats

Les résultats sont présentés sur des graphiques à deux dimensions. L'axe horizontal de ces graphiques représente le nombre de polygones actuellement rendus à travers le point de vue de la caméra par rapport à l'ensemble des polygones existants dans la scène complète, exprimé en pourcentage, et l'axe vertical représente le temps de traitement par image, exprimé en FPS.

Comme nous l'avons mentionné à la section 4.2.4, nous avons retenu cette unité pour l'axe horizontal, car elle permet d'exprimer la valeur absolue des résultats des tests obtenus sur une échelle commune. En effet, puisque la quantité de polygones varie selon les scènes générées, cette échelle permet d'uniformiser l'unité de mesure et permet de comparer facilement les tests réalisés, peu importe le nombre de sommets qui constituent la scène 3D. D'autre part, bien qu'exprimée en FPS, l'unité de l'axe vertical est le temps de traitement par image pour les raisons expliquées à la section 4.2.2 de ce document. L'objectif derrière ces choix d'unité de mesure est d'établir une corrélation entre la quantité d'information utilisée pour produire une image et la vitesse d'affichage de celle-ci.

Chacun des graphiques présentés dans cette section vise à illustrer l'ampleur de l'impact sur les *performances d'un système* selon la variation d'un unique paramètre graphique influant la *quantité d'information* rendue à l'écran. Pour chacun des tests effectués, nous avons donc fait varier un seul paramètre influant la qualité/performance graphique à la fois et mesuré en sortie

l'impact que celui-ci a eu sur les performances de traitement graphique du système. Chacun de ces impacts sera analysé et discuté à la section 6.2.

Pour valider l'efficacité de notre banc de tests, nous avons exécuté celui-ci sur les trois plateformes de capacités/performances différentes présentées à la section 4.3. Premièrement, nous avons lancé l'Outil sur un prototype de plateforme avionique de nouvelle génération de notre partenaire industriel. En plus de récolter des résultats sur cette plateforme, l'objectif était, aussi de valider notre banc de tests en le comparant avec leur application SVS présentée à la section Chapitre 3. Cette étape est considérée comme déterminante pour l'aboutissement de notre étude, car on doit confirmer les résultats générés par le banc de tests par rapport à une *référence*. Deuxièmement, nous avons exécuté l'Outil sur une plateforme embarquée de faible puissance nommée BeagleBoard-xM. Finalement, un ordinateur de bureau conventionnel représentera, d'un côté, un système où les performances ne seront pas bridés et, de l'autre côté, l'environnement de développement réel que les concepteurs de la plateforme auront à leur disposition pour optimiser les performances.

Nous avons réalisé ces tests sur ces plateformes additionnelles afin de vérifier que les impacts sur les particularités décelés sur une plateforme ne soient pas propres à celle-ci et qu'il soit possible d'en dégager une tendance ou une similitude pour l'ensemble des systèmes graphiques.

Nous vous rappelons que les détails techniques des ressources matérielles nous sont inconnus. Cette réalité nous oblige à expérimenter chaque paramètre suspecté de contribuer à la variation des performances indépendamment et tirer des conclusions utilisables par les architectes des futurs systèmes utilisant cette ressource. Cette heuristique basée sur les « essais et mesures » nous est imposée par les manufacturiers qui tentent de protéger leurs propriétés intellectuelles. Cependant, les mesures à la base de nos observations réalisées nous permettent de tirer des conclusions utiles pour : (1) estimer les effets de différents paramètres pouvant avoir un impact sur l'optimisation des performances, sur la qualité graphique ou sur les fonctionnalités intégrées, (2) comparer des processeurs graphiques entre eux, et (3) estimer la charge que représente une application graphique sur la plateforme matérielle choisie.

Voici maintenant les descriptions matérielles des plateformes utilisées dans cette étude suivies des résultats qui leurs sont associés.

6.1.1 Résultats sur le pré-prototype de plateforme avionique

Cette section présente les résultats obtenus pour chacun des tests de l'Outil sur le pré-prototype de plateforme avionique muni de la carte graphique Matrox M9148.

6.1.1.1 Tests sur la résolution des tuiles de terrain :

Le premier test fait varier le nombre de sommets qui définissent une tuile de terrain. Trois valeurs ont été choisies afin d'avoir des tuiles de 9x9, 17x17 et 25x25 sommets par tuile. Le choix de ces valeurs provient de l'application SVS étudiée. Nous nous sommes basés sur les paramètres actuels de l'application pour des fins de référence et avons choisi des valeurs de tests qui varient près de cette valeur. L'application utilise des tuiles constituées de 25x25 sommets. Tous les autres paramètres sont fixés à :

- grille d'une dimension de 17x17 tuiles,
- texture placée sur les tuiles de taille 128x128 pixels RGBA,
- une taille de la fenêtre de projection de 640x480 pixels,
- l'intégralité des tuiles sont à haute résolution (tous les sommets sont utilisés pour définir la scène 3D).

Le graphique illustré à la Figure 45 montre l'évolution du FPS par image selon la portion de la scène rendue à l'écran pour les trois valeurs de sommets par tuile.

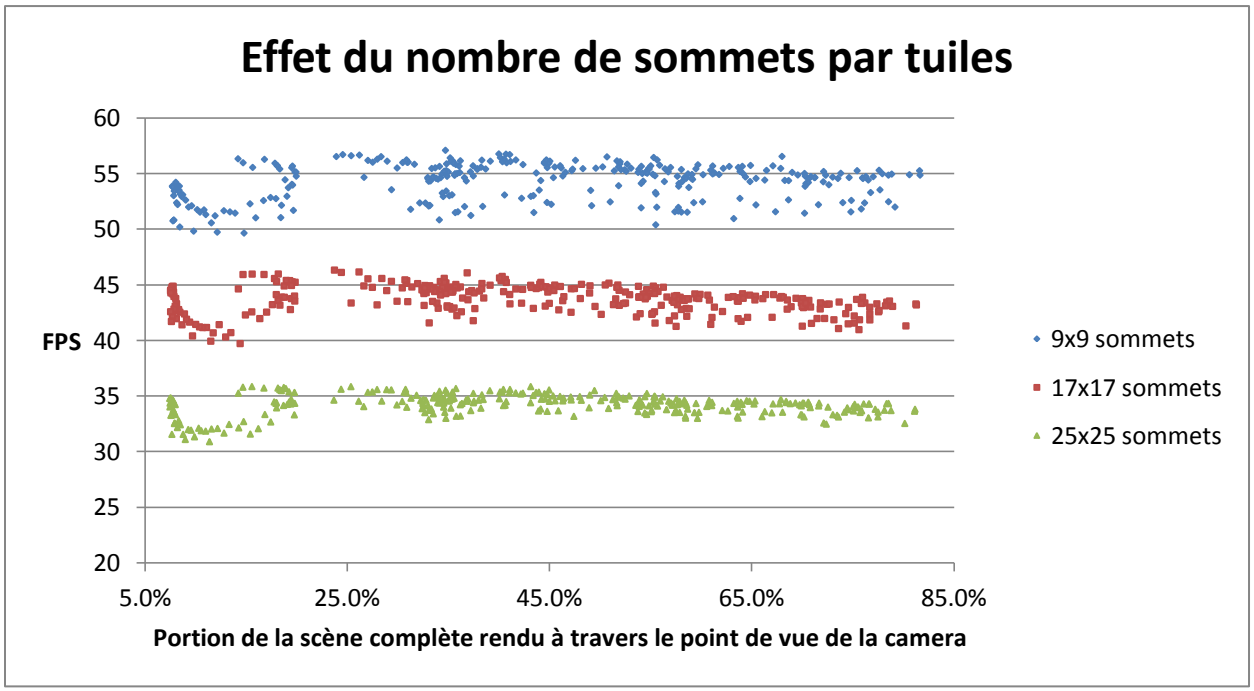

Figure 45 : Effet du nombre de sommets par tuiles selon le pourcentage de la scène rendue

Un deuxième test a été réalisé cette fois-ci afin de mettre l'accent sur la dégradation des performances due à l'augmentation du nombre de sommet par tuile. Le graphique illustré à la Figure 46 montre l'évolution du FPS moyen selon la quantité de polygone qui constitue la scène 3D. Les valeurs testées pour ce test sont les suivantes : 7x7, 9x9, 13x13, 17x17, 21x21, 25x25, 31x31, 37x37 et 45x45 sommets par tuile. Ces valeurs ont été traduites en nombre de polygones constituant la scène pour l'évolution du FPS moyen selon le niveau de détail de la scène. Tous les autres paramètres sont fixes. Chaque échantillon correspond au FPS moyen établi sur l'ensemble des temps de traitement par image d'un test contrairement au test précédent qui calculait le temps de traitement par image.

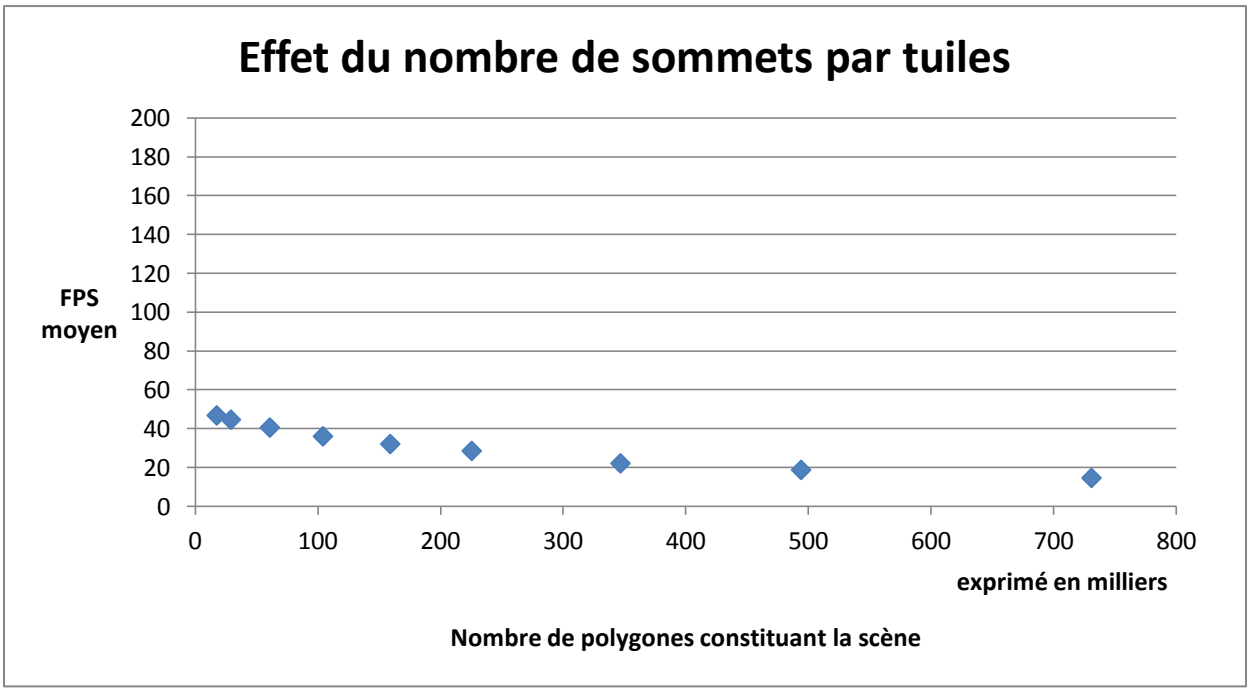

Figure 46 : Effet du nombre de sommets par tuiles

6.1.1.2 Tests sur le nombre de tuiles

Le premier test effectué fait varier la taille de la grille de tuiles qui représente la scène. Trois valeurs ont été choisies de sorte que nous ayons des grilles de terrain de dimension : 9x9, 17x17 et 25x25 tuiles. Ces valeurs sont les mêmes que celles du premier test portant sur l'effet du nombre de sommets par tuile pour avoir le même nombre de sommets qui définissent la scène 3D dans les deux séries de tests. Les autres paramètres sont :

- la résolution d'une tuile est de 17x17 sommets,
- la texture placée sur les tuiles à une taille de 128x128 pixels RGBA,
- la taille de la fenêtre de projection est de 640x480 pixels,
- l'intégralité des tuiles sont à haute résolution (tous les sommets sont utilisés pour définir la scène 3D).

Le graphique illustré à la Figure 47 montre l'évolution du temps de traitement par image (en FPS) selon la portion de la scène rendue à l'écran pour les trois valeurs du test.

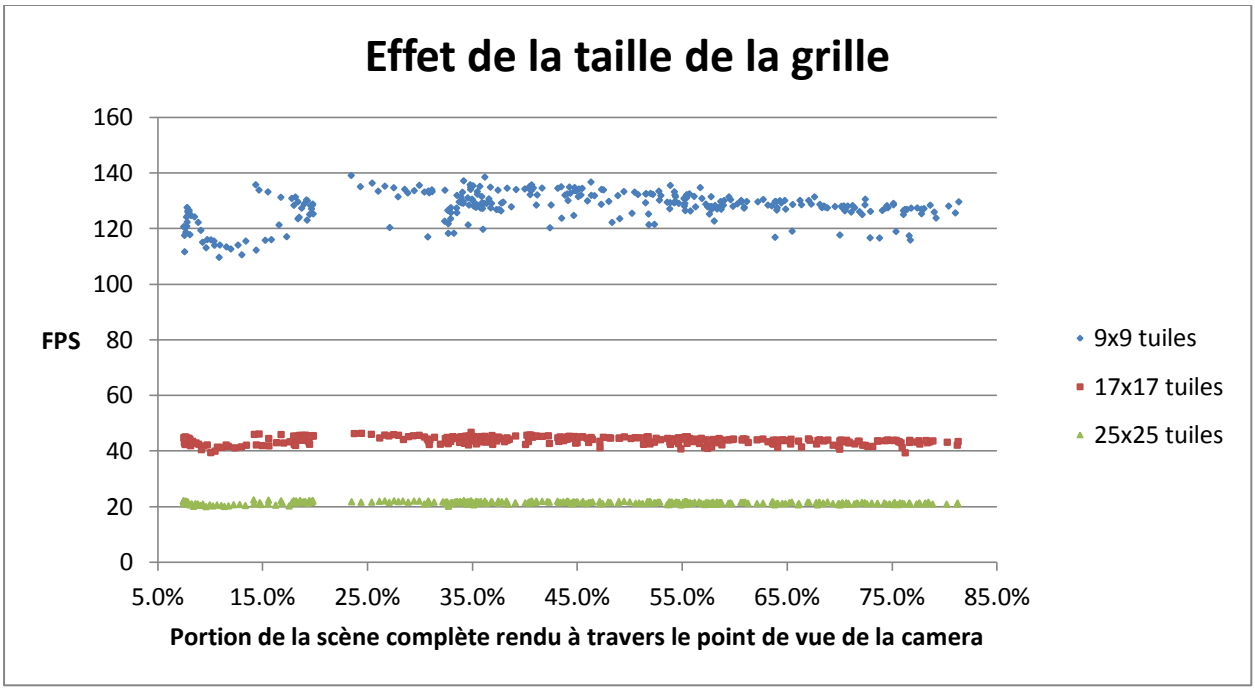

Figure 47 : Effet de la taille de la grille de tuiles selon le pourcentage de la scène rendue

Un deuxième test a été réalisé afin de mettre l'accent sur la dégradation des performances due à l'augmentation du nombre de tuiles de terrain composant la scène. Le graphique illustré à la Figure 48 montre l'évolution du FPS moyen selon la quantité de polygone qui constitue la scène 3D. Les valeurs testées pour ce test sont les suivantes : 7x7, 9x9, 13x13, 17x17, 21x21, 25x25, 31x31, 37x37 et 45x45 tuiles. Encore une fois, tous les autres paramètres sont fixes et chaque échantillon correspond au FPS moyen établi sur l'ensemble des temps de traitement par image d'un test.

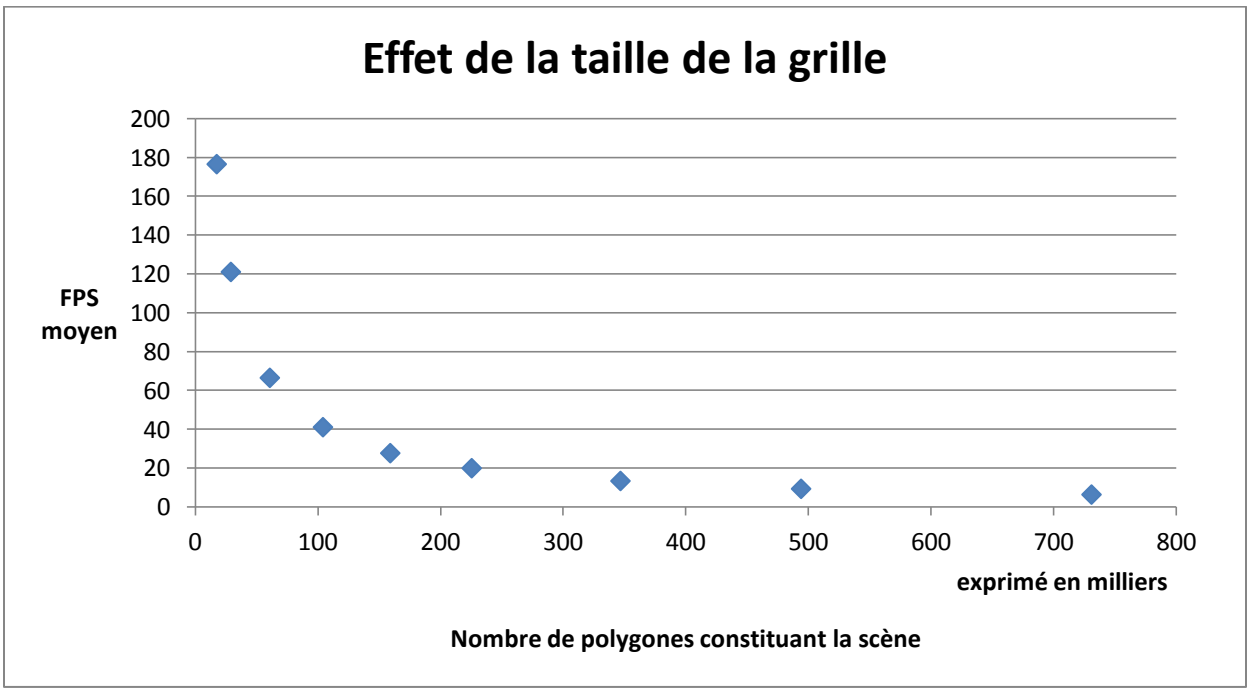

Figure 48 : Effet de la taille de la grille de tuiles

6.1.1.3 Tests sur la résolution de la fenêtre de projection

Ce test fait varier la résolution de la fenêtre de projection. Trois valeurs ont été choisies afin d'obtenir des fenêtres avec les résolutions suivantes : 640x480, 1024x768 et 1280x960 pixels, tous des formats 4:3. Notons que ce format représente le format usuel dans l'industrie aéronautique [42]. Tous les autres paramètres sont fixes pour la réalisation de ce test :

- la grille de tuiles a une dimension de 17x17 tuiles,
- les tuiles de terrain sont composées de 17x17 sommets,
- la texture placée sur les tuiles à une taille de 128x128 pixels RGBA,
- l'intégralité des tuiles sont à haute résolution.

Le graphique illustré à la Figure 49 montre l'évolution du FPS par image selon la portion de la scène rendue à l'écran pour les trois valeurs du test.

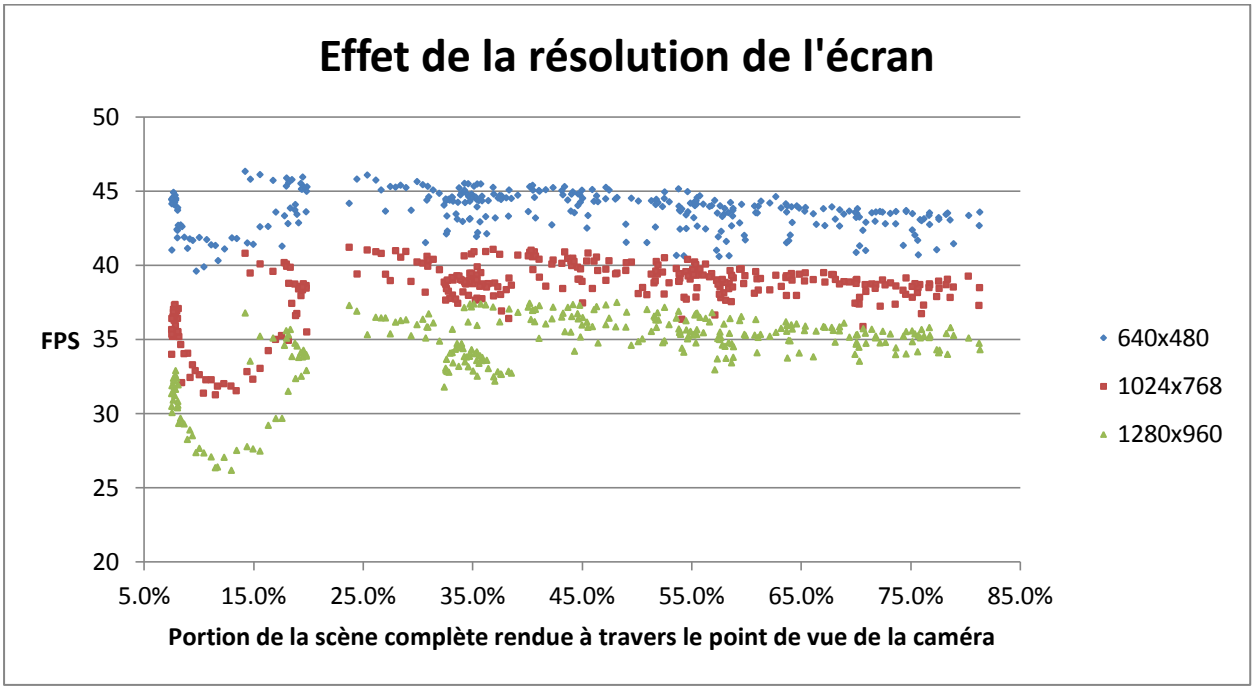

Figure 49 : Effet de la résolution de la fenêtre d'affichage selon le pourcentage de la scène rendue

Un deuxième test a été réalisé afin de bien visualiser l'impact sur les performances obtenues par la variation de la résolution de la fenêtre d'affichage. Le graphique illustré à la Figure 50 montre l'évolution du FPS moyen selon la résolution de la fenêtre d'affichage. Les valeurs utilisées pour ce test sont : 640x480, 800x600, 1024x768, 1152x864 et 1280x960. Cette plage de résolutions couvre toutes les résolutions en format 4:3 standards qu'il était possible d'afficher sur la plateforme cible. Tous les autres paramètres sont fixes et chaque échantillon correspond au FPS moyen établi sur l'ensemble des temps de traitement par image d'un test.

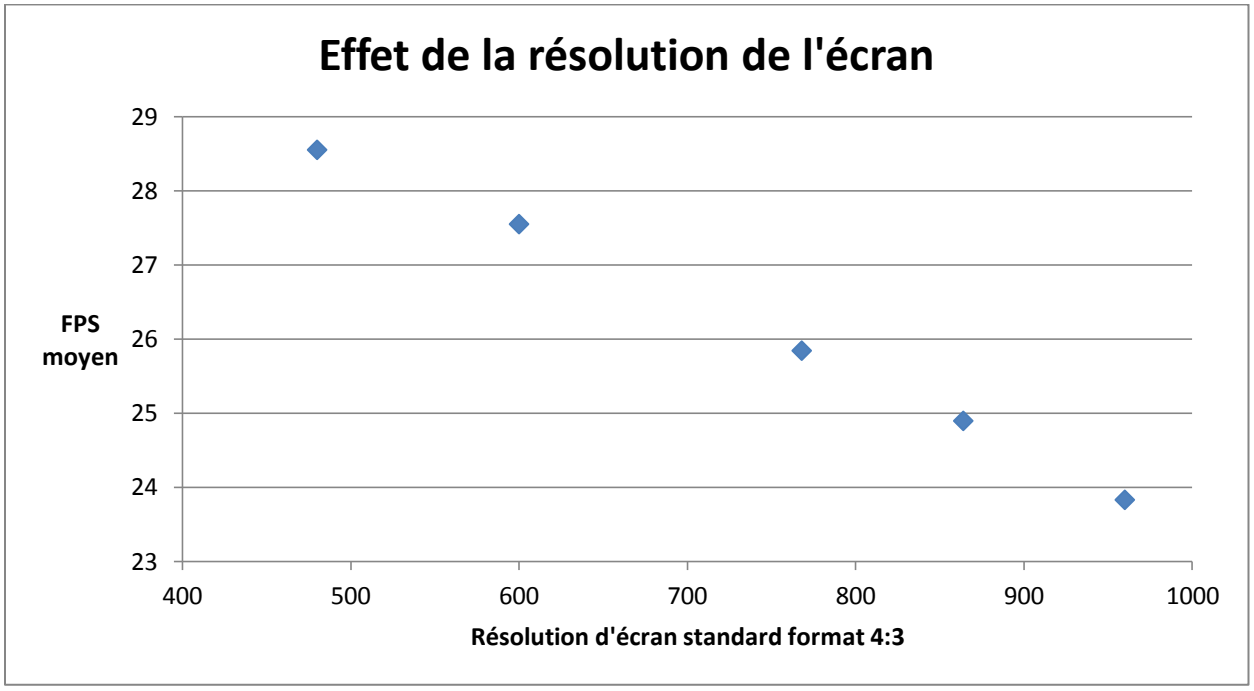

Figure 50 : Effet de la résolution de la fenêtre d'affichage

6.1.1.4 Test sur la taille de la texture des tuiles

Ce test fait varier la taille de la texture qui couvre les tuiles de terrain. Notons qu'il s'agit de la même texture (ressource) qui est appliquée sur chaque tuile. Les trois valeurs retenues pour ce test sont des textures de dimension : 128x128, 512x512 et 2048x2048 pixels. Le choix de ces valeurs provient de l'application SVS étudiée. Nous nous sommes basés sur les paramètres actuels de l'application pour des fins de référence et avons choisi des valeurs de tests qui varient près de cette valeur. Les autres paramètres resteront fixes :

- la grille de tuiles a une dimension de 17x17 tuiles,
- la résolution d'une tuile de terrain est de 17x17 sommets,
- la taille de la fenêtre de projection est de 640x480 pixels,
- l'intégralité des tuiles sont à haute résolution.

Le graphique illustré à la Figure 51 montre l'évolution du FPS par image selon la portion de la scène rendue à l'écran pour les trois valeurs du test.

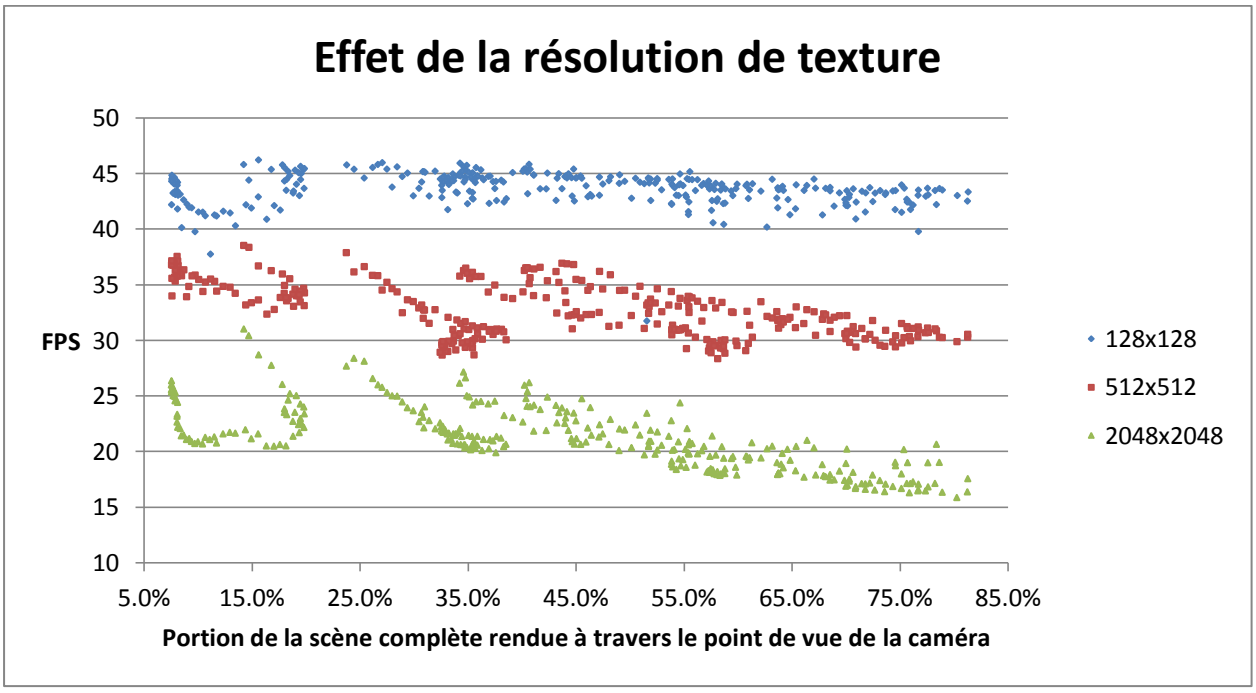

Figure 51 : Effet de la taille de la texture d'une tuile selon le pourcentage de la scène rendue

Un deuxième test a été réalisé afin de bien visualiser l'impact sur les performances obtenues par la variation de la taille de la texture placée sur les tuiles de terrain. Le graphique illustré à la Figure 52 montre l'évolution du FPS moyen selon la taille de texture utilisée pour le test. Les valeurs utilisées sont : 8x8, 16x16, 32x32, 64x64, 128x128, 256x256, 512x512, 1024x1024, 2048x2048 et 4096x4096 pixels. Tous les autres paramètres sont fixes et chaque échantillon correspond au FPS moyen établi sur l'ensemble d'un test.

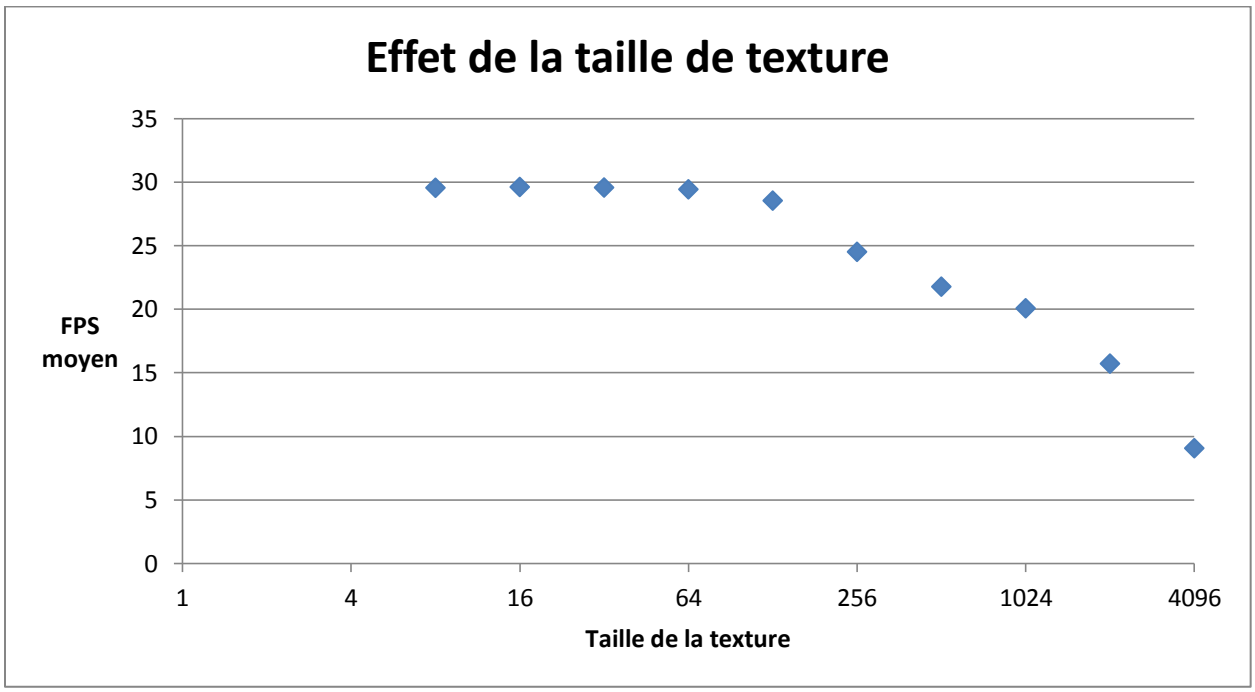

Figure 52 : Effet de la taille de la texture d'une tuile

6.1.1.5 Test sur la profondeur des tuiles à haute résolution

Ce test fait varier la profondeur des tuiles à haute résolution par rapport à la tuile centrale. La tuile centrale est toujours à haute résolution et plus la « profondeur » est grande, plus grand sera le ratio entre le nombre de tuiles de haute résolution et celles, limitrophes, de basse résolution. Trois ratios de profondeur ont été choisis afin de réaliser ce test : 0/2, 1/2 et 2/2. Pour la réalisation du test avec un ratio de profondeur de tuiles à haute résolution de 1/2 (donc avec la moitié (1/2) des tuiles situées entre la tuile centrale et la limite du terrain à haute résolution et les subséquentes à basse résolution) et en se référant au schéma de la Figure 53, seules les tuiles vertes et bleues seront à haute résolution. Par conséquent, un ratio de 0/2 implique que seule la tuile bleue est à haute résolutions et qu'un ratio de 2/2 implique que la totalité des tuiles sont à haute résolution. Tous les autres paramètres sont fixes pour la réalisation de ce test :

- la grille de tuiles a une dimension de 11x11 tuiles,
- la résolution d'une tuile de terrain est de 25x25 sommets,
- la texture placée sur les tuiles a une taille de 128x128 pixels RGBA,
- la taille de la fenêtre de projection est de 640x480 pixels.

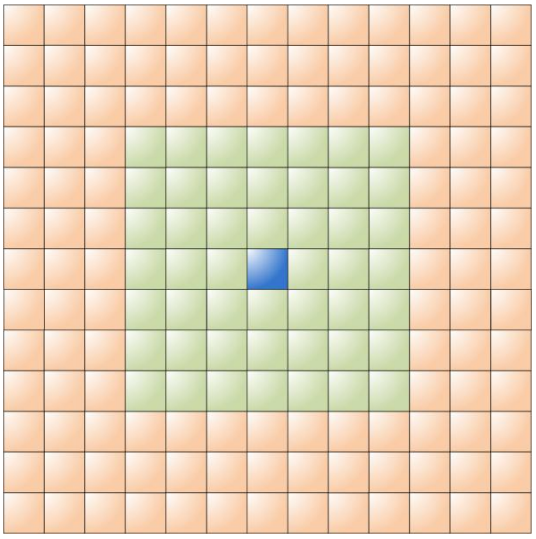

Figure 53 : Répartition des tuiles à haute et basse résolution selon une profondeur donnée

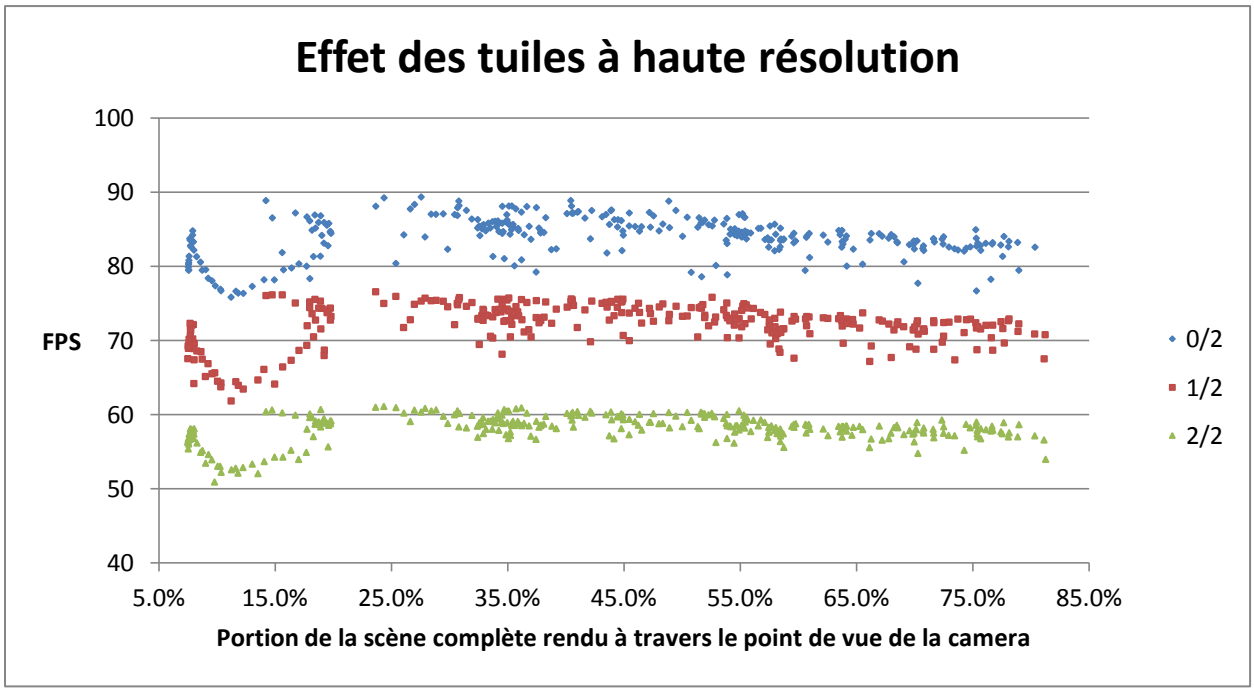

Figure 54 : Effet des tuiles à haute résolution selon le pourcentage de la scène rendue

6.1.1.6 Test sur l'effet du brouillard

Ce test mesure l'impact de l'ajout d'un effet de brouillard sur la scène. Pour effectuer ce test nous avons utilisé une grille de tuiles de dimension 17x17, une résolution de tuile de terrain de 17x17 sommets, une texture de tuile ayant une taille de 128x128 pixels RGBA, une taille de fenêtre de projection de 640x480 pixels et notons que l'intégralité des tuiles sont à haute résolution.

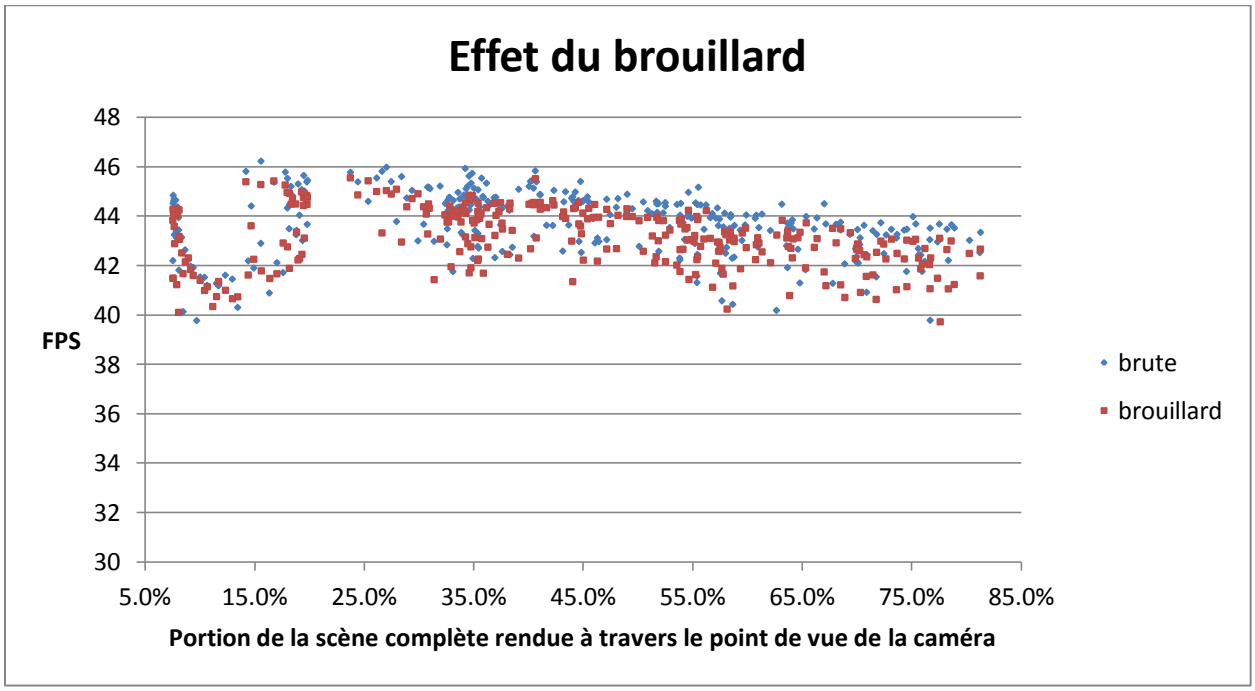

Figure 55 : Effet du brouillard selon le pourcentage de la scène rendue

6.1.2 Résultats sur la plateforme BeagleBoard-xM et l'ordinateur de bureau

Afin de faciliter la lecture de cet ouvrage, nous avons présenté les résultats de ces plateformes en Annexe A.

6.2 Discussion des résultats

Suite aux résultats présentés à la section précédente, nous allons maintenant décrire et analyser les différents comportements que nous avons remarqués. En premier lieu, nous allons valider notre outil avec le prototype d'application SVS de notre partenaire industriel. Nous allons ensuite décrire les dégradations de performance que nous avons observées suite à la variation des paramètres graphiques présentés à la section 4.2.1. Finalement, nous discuterons de l'utilité de notre outil lors d'activités de conception d'application graphique de type génération d'images en temps réel et de l'extensibilité/adaptabilité de celui-ci à d'autres environnements.

6.2.1 Validation des résultats avec le prototype d'application SVS caractérisé

La première étape de notre analyse de résultats consiste à valider l'Outil en le comparant avec l'application SVS de notre partenaire industriel. L'objectif est de confirmer les résultats générés par le banc de tests par rapport à une *référence*. Pour y arriver, nous avons paramétré l'Outil de sorte qu'il reproduise fidèlement la configuration des éléments graphiques contenus dans une scène 3D de l'application SVS. Ensuite nous l'avons lancé sur leur prototype de plateforme avionique de nouvelle génération, cette même plateforme sur laquelle est installée l'application SVS.

Le Tableau 4 présente le FPS moyen de l'application SVS fourni par notre partenaire industriel et le FPS moyen de la simulation effectuée par notre Outil. Ce dernier a évalué un FPS moyen 9,70% inférieur à celui de l'application réelle. Cette différence non négligeable est attribuée aux scénarios de tests de notre Outil. En effet, le FPS moyen de l'application SVS a été obtenu suite à une utilisation réelle de l'application. Bien évidemment, dans un scénario d'utilisation réel, jamais le point de vue de la caméra n'atteindra une altitude à laquelle il serait possible de voir l'intégralité du terrain présent en mémoire. La caméra restera à une altitude plus basse où il est possible d'afficher des images constituées de tout au plus du tiers de la scène totale, approximativement. Nous verrons dans les sections suivantes que ces images moins complexes à produire sont donc plus rapides à traiter. Puisque l'échantillonnage de notre Outil traite des images plus complexes que celles traitées par l'application SVS, il est par conséquent

normal que le taux de FPS moyen mesuré par l'Outil soit moins élevé que celui fourni par notre partenaire industriel (23,7 FPS).

Tableau 4 : Comparaison du FPS moyen entre l'application SVS et l'Outil

	Application SVS commerciale	Outil de mesure synthétique
FPS moyen	23,7 FPS	21,4 FPS

6.2.2 Dégradation des performances liée à la portion de scène rendue

L'intégralité des tests réalisés a montré une dégradation des performances graphiques proportionnelle à la portion de la scène complète rendue à travers le point de vue de la caméra. Outre certains cas d'exceptions relatés dans les prochains paragraphes, cette dégradation est généralement graduelle et proportionnelle à la portion scène affichée à l'écran : plus ce pourcentage est élevé, plus le temps de traitement par image augmente, donc le FPS instantané par image (IFPS) diminue.

Bien que cette variation linéaire des performances soit présente sur chacune des plateformes testées, nous avons remarqué que la sensibilité de cette variation est propre à chaque système. En effet, selon la plateforme utilisée, nous obtenons des variations de performances différentes entre le temps de traitement par image des images à faible pourcentage de scène et le temps de traitement par image des images à fort pourcentage de scène. Toutefois, en se référant aux graphiques qui illustrent les résultats des tests portant sur la quantité de sommets utilisés pour décrire la scène (Figure 45, Figure 47, Figure 56, Figure 58, Figure 67 et Figure 69), nous avons remarqué que ce pourcentage de dégradation reste constant sur une même plateforme. En d'autres termes, les ratios entre les temps de traitement des images à faible pourcentage de scène et celui de celles à fort pourcentage de scène d'une même plateforme restent constants. Donc pour une scène 3D donnée, les variations du FPS associées à cette dernière sont comprises à l'intérieur de ce ratio. Ceci donne, entre autres, cet effet d' « aplatissement » de la courbe mis en évidence à la Figure 69. Le Tableau 5 énumère les ratios de variations de temps de traitement par image obtenus pour les trois plateformes testées. Voici un exemple pour mieux interpréter ces ratios : si

on obtient un temps de traitement par image de 100 IFPS pour une scène et une prise de vue de caméra donnée sur la plateforme avionique, on peut prévoir que le pire cas de temps de traitement par image sur cette même scène, mais avec une prise de vue différente, sera de 92 IFPS.

Tableau 5 : Variations des temps de traitement par image selon la plateforme

	BeagleBoard-xM	**Plateforme avionique**	**Ordinateur de bureau**
Variation du IFPS pour une scène	~62%	~8%	~21%

Selon les valeurs présentées dans le tableau précédent, on constate que les performances graphiques du BeagleBoard-xM sont très sensibles à la portion de scène rendue tandis que celles offertes par la carte graphique Matrox M9148 de la plateforme avionique sont plus « stables ». Dans un contexte avionique où les contraintes relatives au déterminisme et à la fiabilité du système sont omniprésentes, cette continuité, ou faible variation, offre l'avantage de prédire plus facilement les performances du système graphique. Ceci met aussi en évidence la possible présence d'un mécanisme de *clipping*[3] (voir section 5.3.1.2) qui semble très efficace sur le BeagleBoard-xM. Noter que seule cette plateforme embarquée aux ressources limitées utilise la librairie native OpenGL ES 1.4 et que les deux autres utilisent une librairie émulée.

Bien que la dégradation des performances de traitement graphique soit proportionnelle à la portion de la scène complète rendue à travers le point de vue de la caméra, il existe sur les graphiques présentés aux sections 6.1.1 et 6.1.2 plusieurs nuages de points qui dérogent à cette tendance. Ce phénomène est mis de l'avant notamment sur le graphique de la Figure 49. On constate sur cette figure que les images définies par un faible pourcentage de scène (<20%) ont un FPS nettement inférieur à celui proposé par la courbe de tendance. On remarque un phénomène similaire pour les images constituées de 30% à 40% de la scène. En analysant ces

[3] Rappel : le *clipping* consiste en une opération d'élagage qui élimine les polygones situés à l'extérieur du volume de projection (polygones non rendus à l'écran).

images plus gourmandes en puissance de traitement, nous avons remarqué qu'elles correspondaient aux cycles de caméra où cette dernière se trouvait près du sol et où le tampon de profondeur était le plus sollicité. En d'autres termes, les images capturées par la caméra à ces instants sont celles où le compte de pyramides obstruées était le plus élevé. Bien que ce scénario de dénivellation de terrain soit peu probable dans un cas d'utilisation réel, ceci montre l'importance que les concepteurs d'application de type SVS doivent porter aux performances graphiques liées à la surutilisation du tampon de profondeur.

Notons que l'absence d'échantillon entre 20% et 25% est due au fait que nous n'avons pu produire des images avec des prises de mesures correspondantes, en fonction de l'orientation de la caméra.

6.2.3 Dégradation des performances due à la quantité de polygones

Dans la batterie de tests que nous avons lancée, nous avons aussi mesuré l'impact de nombre de polygones servant à décrire une scène 3D. Sans surprise, plus ce nombre est élevé plus le FPS de l'application graphique écope. Sur les graphiques présentés aux Figure 46, Figure 48, Figure 57, Figure 59, Figure 68 et Figure 70, on constate que la dégradation des performances graphiques due au nombre de polygones constituant la scène décroit en suivant une tendance logarithmique, et ce, peut importe la plateforme. Toutefois, ce qui pique la curiosité avec ces tests est de visualiser l'évolution du FPS moyen de ces mêmes graphiques par paire selon leurs plateformes respectives. Prenons par exemple les graphiques des Figure 46 et Figure 48 de la plateforme avionique. Chacun des points du graphique de la Figure 46 est équivalent en termes de quantités de polygones constituants la scène aux points du graphique de la Figure 48. La différence majeure entre les deux ensembles de points est la répartition des polygones dans les objets constituant la scène. Dans le premier cas, le nombre de tuiles de terrain (objets 3D) est fixe et seul le nombre de polygones par tuile varie et, dans le deuxième cas, le nombre de tuiles varie tandis que le nombre de polygones par tuile est fixe. Bien que dans les deux cas on remarque une dégradation qui suit une tendance logarithmique, on remarque aussi qu'à valeur égale en termes de polygones constituant la scène, le nombre d'objets 3D présents sur la scène influence beaucoup plus les performances que le nombre de polygones qui constituent ces mêmes objets. Par conséquent, dans un contexte d'optimisation des performances de traitement graphique d'une

application de type SVS, il vaut mieux privilégier l'augmentation de la superficie couverte par une tuile de terrain tout en lui injectant de nouveaux sommets et minimiser subséquemment le nombre de tuiles qui définissent la scène.

Les tests que nous avons effectués ont aussi couvert le mécanisme de double indexation des tuiles : les tuiles à haute résolution où l'intégralité des sommets est utilisée pour effectuer le rendu de l'objet 3D et les tuiles à basse résolution où seul un sommet sur deux est utilisé pour effectué le rendu. L'intérêt de ce type de fonctionnalité est qu'il permet de détailler avec précision les tuiles se situant près de la position de l'observateur et de représenter les tuiles lointaines avec moins de précision, et ce, en obtenant au final un gain substantiel sur les performances graphiques du système. Les tests que nous avons lancés sur les trois différentes plateformes sont concluants, mais offrent des gains d'ampleurs différentes, voir Figure 54, Figure 64, Figure 75.

Le Tableau 6 présente, selon la plateforme cible, les différents gains de performance graphique atteignable grâce au système de double résolution des tuiles. Ils ont été obtenus en divisant les FPS moyens de tests utilisant une scène avec la totalité des tuiles à haute résolution et une seconde scène utilisant seulement la tuile centrale comme tuile à haute résolution. Nos expérimentations on montré que le gain est linéaire par rapport à la profondeur de tuiles à haute résolution choisie.

Tableau 6 : Augmentation du FPS moyen possible via la double indexation des tuiles

	BeagleBoard-xM	**Plateforme avionique**	**Ordinateur de bureau**
Augmentation possible du FPS moyen via le système de double résolution des tuiles	~67%	~43%	~20%

6.2.4 Effets de la résolution de la fenêtre de projection

Parmi les tests que nous avons lancés, un d'entre eux portait sur la résolution de la fenêtre de projection (ou résolution de l'écran). Comme attendu, plus la résolution est grande, moins rapide seront les performances en vitesse de rendu. Cette astuce est bien connue pour augmenter les performances d'un système graphique. Une résolution plus petite implique moins de pixel à dessiner donc un processus de rastérisation plus rapide [43], d'où le gain en performance. Nous avons toutefois remarqué certaines caractéristiques et comportements liés à cette résolution. Premièrement, nous avons constaté sur les trois plateformes testées que l'augmentation de la résolution d'écran infligeait une dégradation constante du FPS moyen du système graphique selon la résolution d'écran choisie. Les graphiques présentés aux Figure 50, Figure 61 et Figure 72 illustre cette dégradation linéaire du FPS moyen due à l'augmentation de cette résolution. Le Tableau 7 répertorie la variation du FPS moyen par plateforme. Étant donné les contraintes de résolution imposées par la plateforme BeagleBoard-xM, les variations de résolution sont comprises entre la plus petite et la plus grande résolution format 4:3 standard disponible sur cette dernière soit 640x480 et 1152x864. Notons que les variations du FPS moyen par plateforme ont été obtenues en effectuant plusieurs tests de résolutions d'écran avec des paramètres de scène différents.

Tableau 7 : Variation du FPS moyen selon la résolution d'écran

	BeagleBoard-xM	Plateforme avionique	Ordinateur de bureau
Variation de résolution	640x480 à 1152x864	640x480 à 1152x864	640x480 à 1152x864
Variation du FPS moyen	~39%	~13%	~6%

Deuxièmement, nous avons remarqué que l'augmentation de la résolution amplifiait les dégradations associées à l'utilisation de tampon de profondeur. Cette amplification est facilement notable sur le graphique de la Figure 49 pour les mêmes échantillons mentionnés à la section

6.2.2, ceux constitués d'un pourcentage de scène de moins de 20% et compris entre 30% et 40%. On y aperçoit des nuages de points qui s'éloignent de plus en plus de la courbe de tendance : plus la résolution augmente, plus ces points s'éloignent de la courbe. Ce comportement est dû à l'augmentation du nombre de pixels à dessiner. Tel que nous l'avons mentionné à la section 5.3.2.1, des tests de profondeur sont appliqués à chacun des pixels à dessiner lors du rendu d'une image. Donc, si nous avons plus de pixels à dessiner, OpenGL aura plus de tests de profondeur, gourmands en puissance de traitement, à effectuer.

6.2.5 Impacts de la taille de texture

Encore une fois sans surprise, plus la taille de la texture est grande, plus la dégradation des performances du rendu graphique est importante. Sur les graphiques présentés aux Figure 51 (plateforme avionique) et Figure 62 (Beagleboard-xM), qui présentent l'évolution de temps du traitement par image selon la portion de scène rendue à travers le point de vue de la caméra, on remarque que lorsque la taille de la texture augmente, les échantillons du graphique ont tendance à former trois zones distinctes : (1) 5-20%, (2) 25-40% et (3) 37-85%. Les zones 2 et 3 forment des pentes linéaires qui sont de moins en moins prononcées plus la portion de scène à rendre est grande, ce qui est en accord avec les observations discutées précédemment. Mais comment expliquer les phénomènes de dégradation prononcée du temps de traitement par image observé dans les zones (1) et (2)? Rappelons ici que ces deux zones d'échantillonnage correspondent à des prises de vue situées près du sol, d'où la plus petite portion de scène couverte par ces échantillons. Deux caractéristiques graphiques expliquent ce comportement : le tampon de profondeur et le filtrage des textures. Comme nous l'avons expliqué dans les sections précédentes de ce chapitre, le tampon de profondeur est très sollicité pour la création de ces images à faible altitude. Mais il n'explique pas à lui seul la dégradation atypique notée dans ces zones. La fonctionnalité de filtrage des textures ajoute une complexité substantielle au traitement d'image. En OpenGL, cette fonctionnalité permet de grossir la texture placée sur les objets situés à proximité de la caméra et de la réduire pour les objets éloignés de la caméra [23]. Donc, les objets texturés situés près de la caméra prennent plus de temps à dessiner, car l'intégralité de la texture est considérée pour le rendu de l'objet en plus d'être filtrée pour diminuer l'effet de pixellisation du au grossissement de celle-ci. À l'inverse, la texture des objets lointains est quant

à elle réduite donc moins de texels sont pris en considération pour dessiner ces objets. C'est pour cette raison que la dégradation du temps de traitement par image est plus « homogène » dans la zone (3). L'ordinateur de bureau a lui aussi montré ces caractéristiques, mais elles sont beaucoup moins marquées (voir Figure 73).

D'autre part, selon les graphiques illustrés aux Figure 52, Figure 63 et Figure 74, la dégradation des performances est limitée jusqu'à une taille de texture de 128x128 texels. Pour les tailles subséquentes, il s'ensuit une dégradation linéaire des performances du rendu graphique. Bien que cette dégradation soit linéaire, nous n'avons pu établir de corrélation entre le ratio de dégradation lié aux tailles de texture et la complexité de la scène à rendre. Par exemple, la scène utilisée pour réaliser l'échantillonnage du graphique de la Figure 52 montre une dégradation de ~68% entre l'utilisation d'une texture de 128x128 et 4096x4096 texels. Ce ratio varie selon la complexité de la scène d'où l'impossibilité d'en dégager une tendance. Toutefois, nous avons noté que ce taux de dégradation reste substantiel sur la plateforme avionique et le BeagleBoard-xM, contrairement à l'ordinateur de bureau muni d'une carte graphique dernier cri. Notons ici que les cartes graphiques nouvelles générations offrent une fonctionnalité de texturisation qui impacte peu sur les performances graphiques d'un système [43].

6.2.6 Impacts de l'effet de brouillard et d'anticrénelage

Suite aux expérimentations menées sur les trois différentes plateformes, on remarque que l'ajout d'effet de brouillard sur la scène 3D affecte très peu les performances graphiques du système. Lorsqu'on se réfère aux graphiques portant sur l'effet du brouillard des Figure 55, Figure 65 et Figure 76, on aperçoit que l'échantillonnage du temps de traitement par image d'une scène avec présence de brouillard suit de très près celui d'une scène sans effet de brouillard. Rappelons ici qu'il s'agit d'un effet de brouillard dynamique dont la profondeur est recalculée pour chaque image (voir section 5.3.2.4), ce qui peut expliquer en partie cette légère dégradation des performances du rendu graphique. De plus, si on se réfère au cas de l'ordinateur de bureau (Figure 76), on remarque que les images à faible portion de scène s'affichent plus rapidement avec l'effet de brouillard. Selon [23], l'utilisation du brouillard dans les applications graphiques peut améliorer dans certains cas les performances de rendu graphique d'un système. En effet, dans les nouvelles versions d'OpenGL, l'engin graphique peut lui-même faire l'optimisation

d'omettre le rendu d'un objet 3D si celui-ci est complètement masqué par le brouillard (moins d'objets à rendre implique un rendu plus rapide). C'est exactement ce qui se produit dans le cas présent. Puisque la profondeur du brouillard est définie par un rayon de longueur fixe autour du point de vue de la caméra, les tuiles situées dans les coins de la grille sont complètement masquées par le brouillard et par conséquent jamais dessinées (d'où ce léger gain de performance).

Le dernier test que nous avons lancé sur les plateformes portait sur l'ajout d'une fonctionnalité d'anticrénelage. Seules les librairies graphiques du BeagleBoard-xM et de l'ordinateur de bureau supportaient l'anticrénelage par multi-échantillonnage, méthode présentée à la section 5.3.2.3. Il nous était donc impossible de lancer ce test sur la plateforme avionique. Les graphiques des Figure 66 et Figure 77 présentes l'impact de cette fonctionnalité sur les performances graphiques des systèmes. On remarque sur le BeagleBoard-xM que l'anticrénelage affecte substantiellement le temps de traitement par image (une diminution de ~18% du FPS moyen). L'impact est cependant moins marqué sur l'ordinateur de bureau (une diminution de ~6% du FPS moyen). Cette diminution s'explique par le multi-échantillonnage des pixels. Chaque pixel a été multi-échantillonné en quatre « mini pixels » pour calculer la couleur finale du pixel, deux à l'horizontale et deux à la verticale. Ceci implique que l'on double virtuellement la résolution d'écran. On obtient ainsi quatre fois plus de pixels à traiter d'où l'augmentation du temps de traitement par image.

6.2.7 Utilité de l'Outil dans la conception d'un système graphique

Comme nous l'avons mentionné un peu plus tôt dans ce document, l'un des objectifs principaux de ce travail était de fournir aux concepteurs d'application graphique avionique un outil d'aide à la conception qui pouvait les assister dans leurs activités de mitigation des risques liées aux performances de traitement graphique d'un système. Les mesures obtenues suite aux expérimentations menées dans cette étude nous ont démontré que l'Outil remplissait cette tâche par rapport à la prédiction de ces performances.

Avec l'Outil, un concepteur d'application graphique avionique peut analyser plusieurs GPU et s'assurer de leurs capacités de traitement graphique. Il peut prédire comment un système de

rendu graphique réagira dans les pires cas d'utilisation et aussi déterminer lequel des produits offre le rendement voulu, et ce, au meilleur prix. D'autre part, l'Outil permet de profiler les performances d'un système graphique selon chacune des caractéristiques graphiques énumérées à la section 4.2.1. Cette fonctionnalité indique l'impact sur les performances dû à chacun de ces paramètres. En sachant ces impacts et leur ampleur respective, un concepteur peut facilement repérer quel(s) paramètre(s) changer pour modifier les performances de traitement graphique de son application. Finalement, l'Outil permet aux concepteurs de valider rapidement leur application graphique afin de pouvoir pivoter si nécessaire.

CHAPITRE 7
CONCLUSION ET TRAVAUX FUTURS

Dans un contexte où l'industrie aéronautique ne cesse d'intensifier le développement de nouvelles fonctionnalités visuellement attrayantes et où le temps de mise en marché doit être le plus court possible, l'étalonnage rapide des performances de traitement graphique dans un environnement avionique certifié devient un enjeu important. Actuellement, aucun outil de profilage des performances graphiques dédié aux architectures embarquées critiques n'a encore été proposé. Il nous est donc venu l'idée d'implémenter un banc de test spécialisé qui serait une contribution adaptée et efficace à cette problématique.

Le premier objectif de ce mémoire était de prouver qu'il est possible de réaliser des applications graphiques hautes performances dans un contexte avionique certifiable qui utilisent des composantes matérielles et logicielles COTS. Le deuxième objectif était de concevoir un outil spécialisé qui permet d'étalonner avec précision la puissance de traitement graphique requis par une application avionique graphique haute performance sur une plateforme ciblée. L'étude que nous avons menée était basée sur un prototype d'application SVS destiné à être porté sur une plateforme avionique.

En conclusion, nous avons développé une méthodologie de profilage du traitement graphique d'un système qui s'inscrit dans le flot de conception d'une application graphique dans un contexte avionique. Cette méthodologie, décrite dans ce mémoire, a donné place à la réalisation d'un outil spécialisé qui a pour mission d'aider les concepteurs d'applications graphiques avioniques dans leurs activités de mitigation des risques. Cet outil paramétrable et multiplateforme permet d'analyser plusieurs produits et de s'assurer de leurs capacités de traitement graphique en plus d'établir le budget en ressources graphiques sur une architecture donnée. Également, l'outil a été développé de façon modulaire, ce qui lui permet d'être facilement adapté pour supporter le profilage d'application graphique, de type synthétisation de terrain, muni de nouveaux éléments graphiques. Grâce à cet outil automatisé, les concepteurs d'application graphique peuvent valider rapidement les performances en traitement graphique

qu'auront leur application sur une plateforme cible et pivoter si nécessaire afin de se conformer aux contraintes de performance auxquelles ces dernières sont soumises.

7.1 Travaux futurs

En ce qui concerne les travaux futurs, il serait fort intéressant de mesurer le temps de chargement des données de la RAM générique vers la RAM GPU. Par exemple, en sachant le temps de chargement d'une tuile de terrain (incluant les sommets, l'index des sommets, les couleurs des sommets et les vecteurs normaux des sommets) un développeur d'application graphique pourrait facilement trouver le meilleur compromis entre la surface de terrain réel couvert par une tuile et le nombre de tuiles (objets 3D) qui composent la scène synthétique. Des tuiles de terrain couvrant une plus grande surface impliquent un ensemble de données plus volumineux par tuile, mais une grille de terrain avec moins de tuiles (le nombre de sommets total restant inchangé). Tel que discuté à la section 6.2.3, le nombre d'objets 3D influence grandement les performances de traitement graphique d'un système.

D'autre part, un certain raffinement du modèle de terrain synthétique pourrait être apporté. En ajoutant des obstacles 3D sur la scène et des repères tels que des pistes d'atterrissage, cela ajouterait à la précision des mesures de temps de traitement par image. Il serait particulièrement intéressant de simuler des zones densément peuplées avec beaucoup d'artéfacts humains (obstacles) et de mesurer l'effet sur les performances graphiques de cet ajout massif d'objets 3D simples concentré dans une zone restreinte. L'objectif ici serait de déterminer les pires scénarios d'utilisation afin de s'assurer qu'au moment opportun le système sur lequel est exécutée l'application possède les ressources nécessaires pour accomplir les tâches de traitement graphique conformément aux contraintes de temps réel préalablement définies.

BIBLIOGRAPHIE

1. Wikipedia. (15/07/2014) *Planche de bord tout écran*. Available from: http://fr.wikipedia.org/wiki/Planche_de_bord_tout_%C3%A9cran. Accédé
2. Dutton, M. and D. Keezer. *The challenges of graphics processing in the avionics industry*. in *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*. 2010.
3. Snyder, M. *Synthetic vision systems - from sandbox to reality*. in *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*. 2004.
4. Wikipedia. (15/07/2014) *Helmet-mounted display*. Available from: http://en.wikipedia.org/wiki/Helmet-mounted_display.
5. Hilderman, V. and L. Buckwalter, *Avionics Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware)*. 2007: Avionics Communications Inc. 244.
6. Moller, R. *State-of-the-Art 3D Graphics for Embedded Systems*. in *Devices, Circuits and Systems, Proceedings of the 6th International Caribbean Conference on*. 2006.
7. (15/07/2014) Avionics, E. *IGL: High performance and efficient OpenGL SC software rasterizer*. Available from: http://www.ensco.com/idatavs/igl.htm.
8. Fantini, G.N.C. and C.A.P.D.S. Martins. *A configurable and portable benchmark for 3D graphics*. in *Computer Graphics and Image Processing, 2001 Proceedings of XIV Brazilian Symposium on*. 2001.
9. United States. Federal Aviation, A., *Synthetic vision and pathway depictions on the primary flight display [electronic resource]*. United States. Federal Aviation Administration. Advisory circular ; AC 23-26., ed. A. United States. Federal Aviation. 2005, [Washington, D.C.]: U.S. Dept. of Transportation, Federal Aviation Administration.
10. Schiefele, J., et al. *Safety relevant navigation and certifiable databases for 3D synthetic vision systems*. in *Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*. 1998.
11. Joncas, D., *COTS GPU Selection Considerations for Mil-Aero Electronics.* Engineers' Guide to Military & Aerospace Technologies, 2012: p. 36-39.
12. (CAST), F.C.A.S.T., *Position Paper CAST-29: Use of COTS Graphical Processors (CGP) in Airborne Display Systems*. 2007.
13. Fulton, R., *RTCA/DO-254 Commercial-off-the-shelf graphical processors usage in airborne display systems*. SoftwAir Assurance Inc.
14. Group, K., *OpenGL SC: Safety-Critical Profile Specification*. 2009.
15. Cole, P. *OpenGL ES SC - open standard embedded graphics API for safety critical applications*. in *Digital Avionics Systems Conference, 2005. DASC 2005. The 24th*. 2005.
16. Beeby, M., *COTS OpenGL Software Eases Certifiable Avionics Code.* COTS Journal, 2001.
17. Rossignol, V., *Optimized Safety-Critical Embedded Display Development with OpenGL SC.* SAE Int. J. Aerosp, 2010: p. 91-94.

18. Wain, R. and C.f.t.C.L.o.t.R. Councils, *Meaningful Benchmarks for Scientific Visualization*. 2007: Council for the Central Laboratory of the Research Councils.

19. Rightware. (15/07/2014) *Benchmarking Software*. Available from: http://www.rightware.com/benchmarking-software/.

20. (15/07/2014) Corporation, S.P.E. *SPEC Membership*. 2013; Available from: http://www.spec.org/spec/membership.html.

21. Ho, S., *OpenGL Performance Evaluation on Multiple Computer Platforms*. 2001: Army Research Laboratory.

22. Corporation, S.P.E., *What is This Thing Called "SPECviewperf®"?* 2007.

23. Shreiner, D., *OpenGL Programming Guide: The Official Guide to Learning OpenGL (7th Edition)*. July 21 2009: Addison-Wesley Professional. 936.

24. Ertem, M.C. *An airborne synthetic vision system with HITS symbology using X-Plane for a head up display*. in *Digital Avionics Systems Conference, 2005. DASC 2005. The 24th*. 2005.

25. Wikipedia. (15/07/2014) *Joint Strike Fighter program*. Available from: http://en.wikipedia.org/wiki/Joint_Strike_Fighter_program.

26. Carroll, K. *Deploying C++ For Use In International Safety-Critical Applications*. in *Systems and Software Technology Conference*. 2007. Tampa Bay, FL.

27. Martin, L., *Air Vehicle C++ Coding Standards For The System Development And Demonstration Program*. 2005.

28. Group, K., *The OpenGL Graphics System: A Specification (Version 1.5)*. 2003.

29. Nakhoon, B. and L. Hwanyong. *A cost-effective OpenGL SC solution for the consumer electronics market: An emulation library approach*. in *Consumer Electronics (ICCE), 2012 IEEE International Conference on*. 2012.

30. OpenGL.org. (15/07/2014) *OpenGL Performance Wiki*. Available from: http://www.opengl.org/wiki/Performance.

31. (15/07/2014) Weatherstone, R. *GPU Reviews: Why Frame Time Analysis is important*. 2013; Available from: http://www.vortez.net/articles_pages/frame_time_analysis,1.html.

32. Antonuk, L.E. and R.A. Street, *Thin-film, flat panel, pixelated detector array for real-time digital imaging and dosimetry of ionizing radiation*. 1993, Google Patents.

33. Limited, I.T. (15/07/2014) *PowerVR SDK*. Available from: http://community.imgtec.com/developers/powervr/graphics-sdk/.

34. Nakhoon, B. and B. Gwang Jin. *Design of OpenGL SC emulation library over the desktop OpenGL 1.3*. in *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*. 2010.

35. beagleboard.org. (15/07/2014) *BeagleBoard-xM Specifications*. Available from: http://beagleboard.org/Products/BeagleBoard-xM.

36. Instrument, T. (15/07/2014) *DM3730, DM3725 - Digital Media Processors*. 2011; Available from: http://www.ti.com/lit/ds/sprs685d/sprs685d.pdf.

37. Limited, I.T. (15/07/2014) *PowerVR Series5 GPU*. Available from: http://www.imgtec.com/powervr/series5.asp.

38. *The Angstrom Distribution*. (15/07/2014) Available from: http://www.angstrom-distribution.org/.

39. (15/07/2014) Group, K. *Creating open standard APIs to enable the authoring and playback of rich media on a wide variety of platforms and devices*. 2014; Available from: http://www.khronos.org/.

40. (15/07/2014) Group, K. *OpenGL Measuring Performance Wiki*. 2012; Available from: http://www.opengl.org/wiki/Performance.

41. Wikipedia, *Triple product.*

42. Spitzer, C.R., *Avionics: Elements, Software and Functions*. 2006: Taylor & Francis.

43. Paul, B. *OpenGL and Window System Intergration*. in *SIGGRAPH '97*. 1997. Los Angeles, California.

ANNEXE A : RÉSULTATS SUR LES AUTRES PLATEFORMES

Résultats sur la plateforme BeagleBoard-xM

Tous les paramètres de tests pour cette plateforme sont les mêmes que ceux utilisés sur le prototype de plateforme avionique à la section 6.1.1.

Tests sur la résolution des tuiles de terrain :

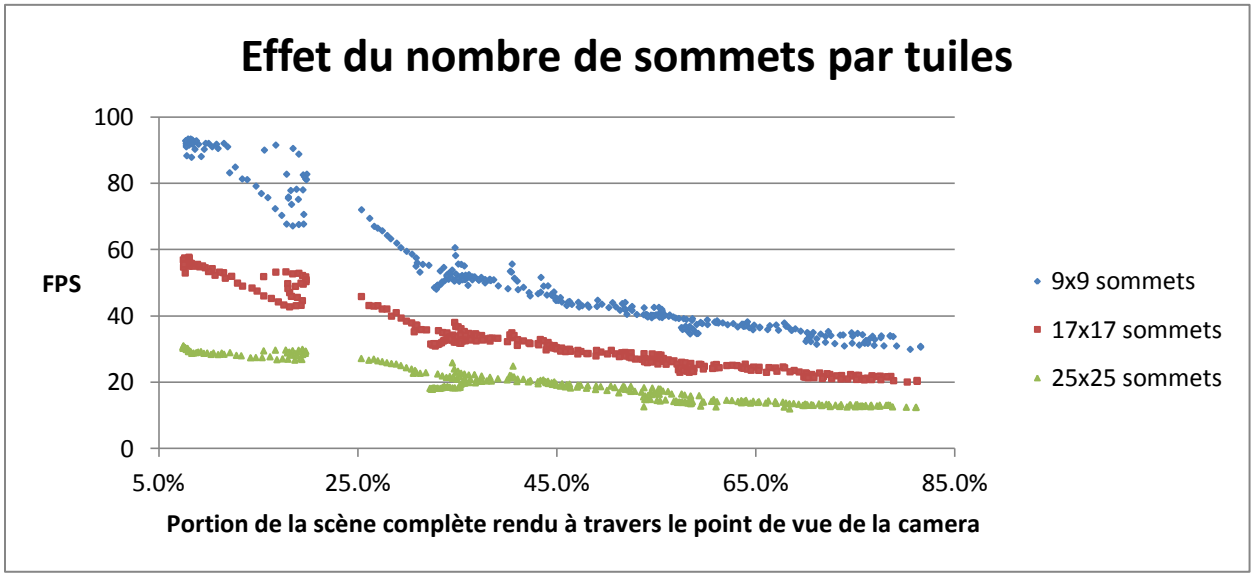

Figure 56 : Effet du nombre de sommets par tuiles selon le pourcentage de la scène rendue

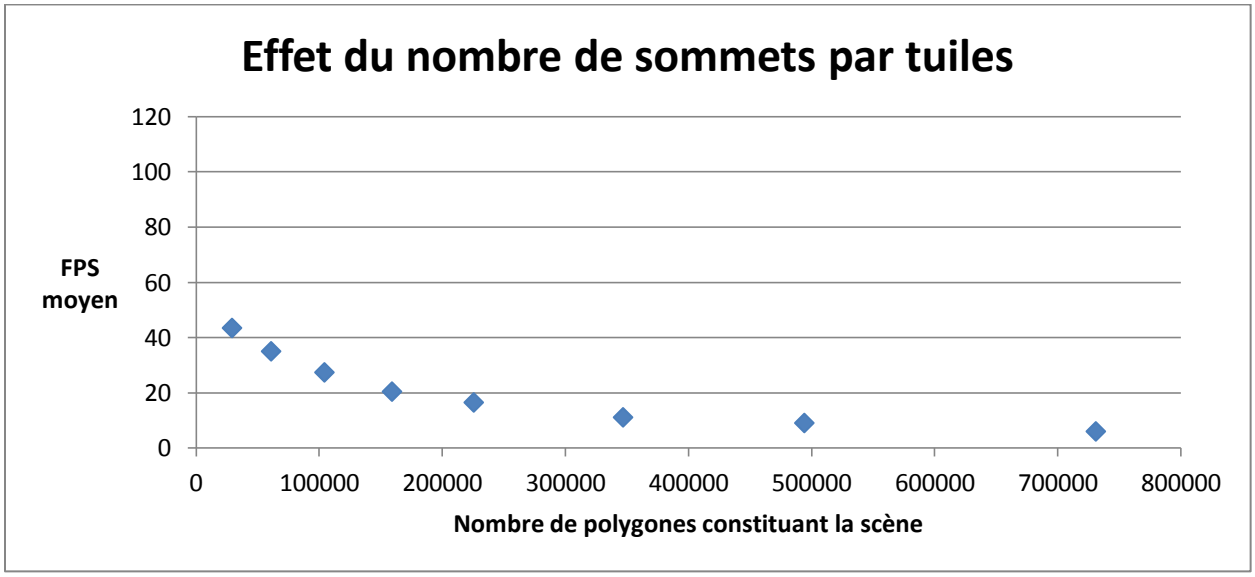

Figure 57 : Effet du nombre de sommets par tuiles

Tests sur la taille de la grille de tuiles

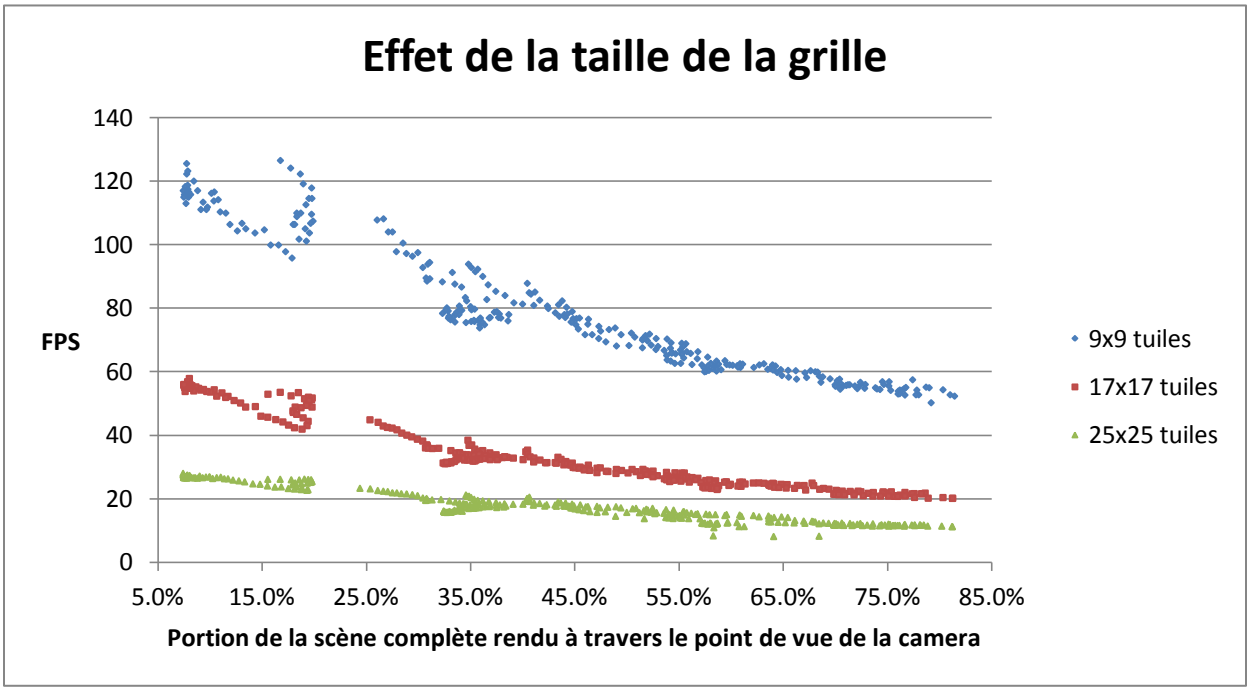

Figure 58 : Effet de la taille de la grille de tuiles selon le pourcentage de la scène rendue

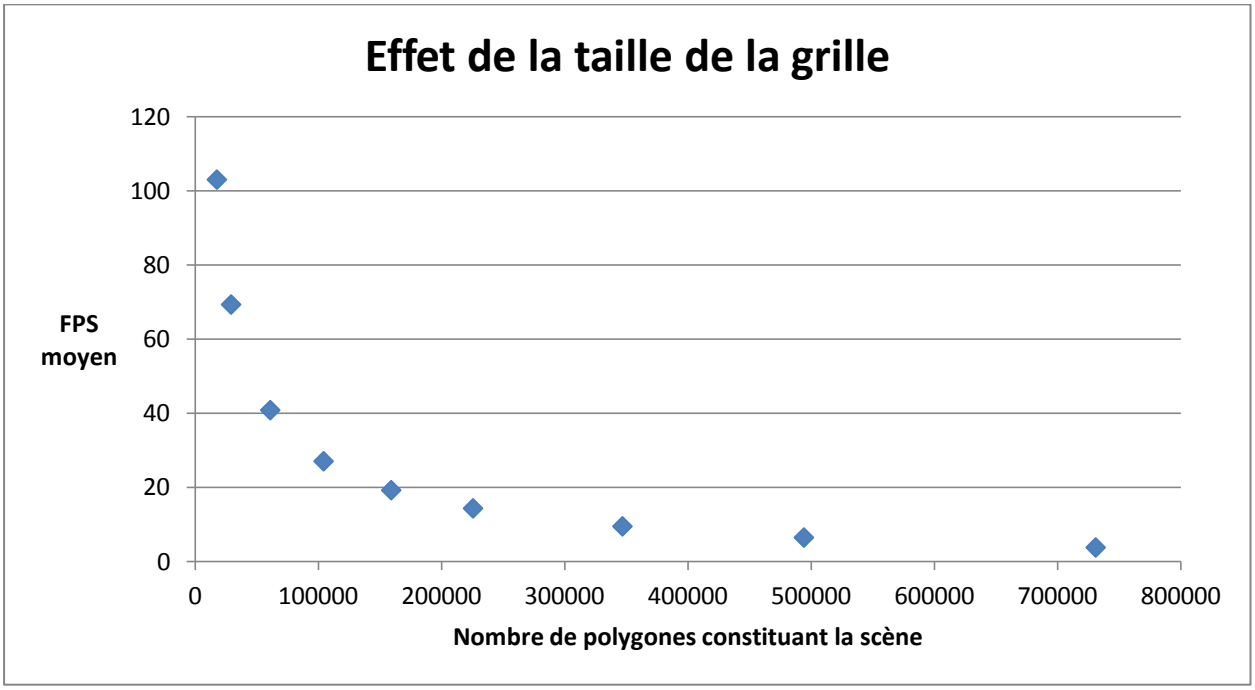

Figure 59 : Effet de la taille de la grille de tuiles

Tests sur la résolution de la fenêtre de projection

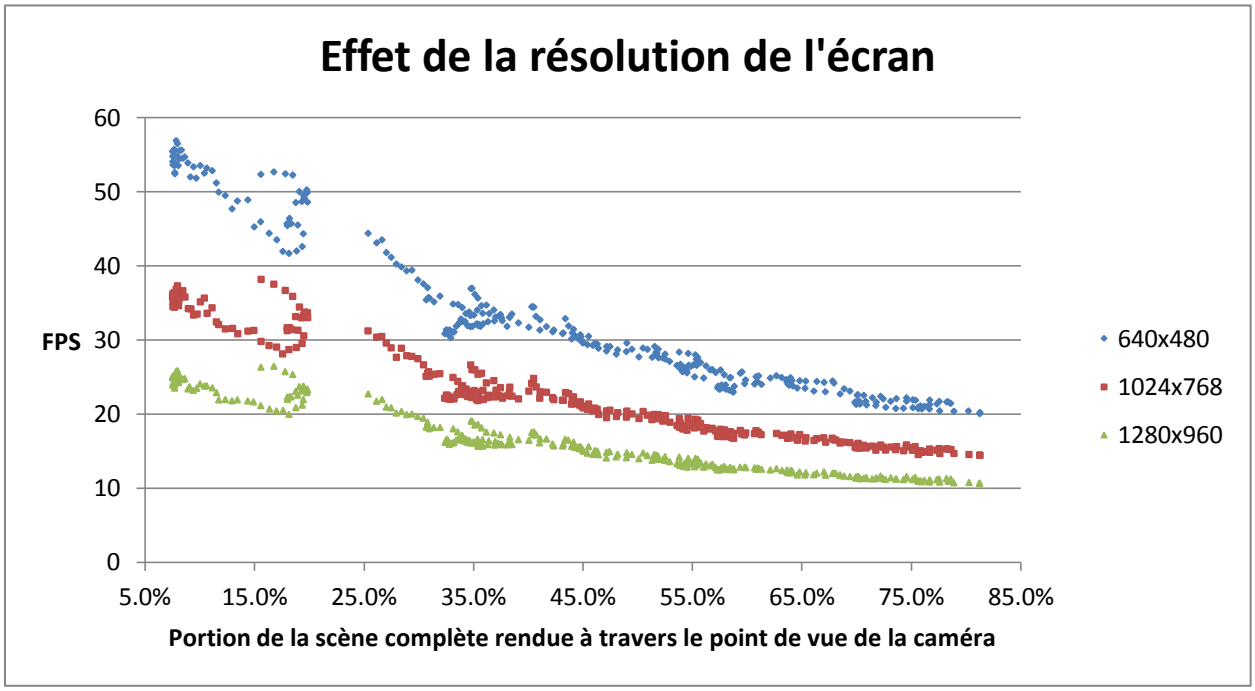

Figure 60 : Effet de la résolution de la fenêtre d'affichage selon le pourcentage de la scène rendue

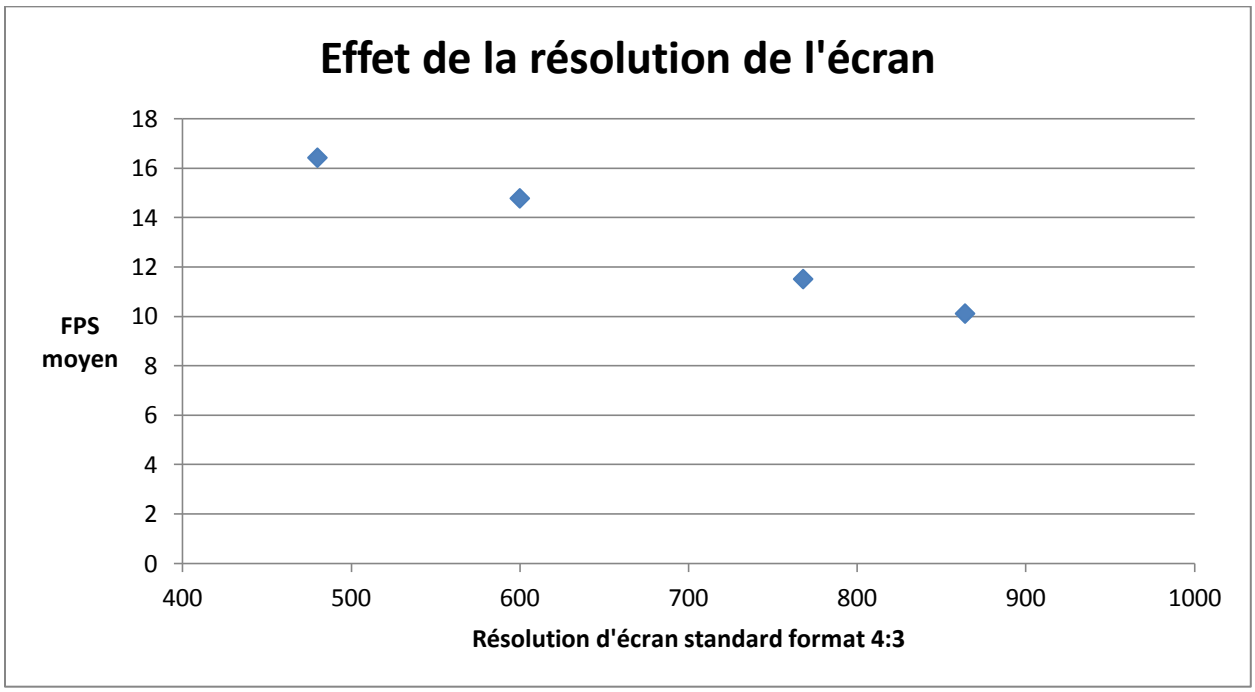

Figure 61 : Effet de la résolution de la fenêtre d'affichage

Test sur la taille de la texture des tuiles

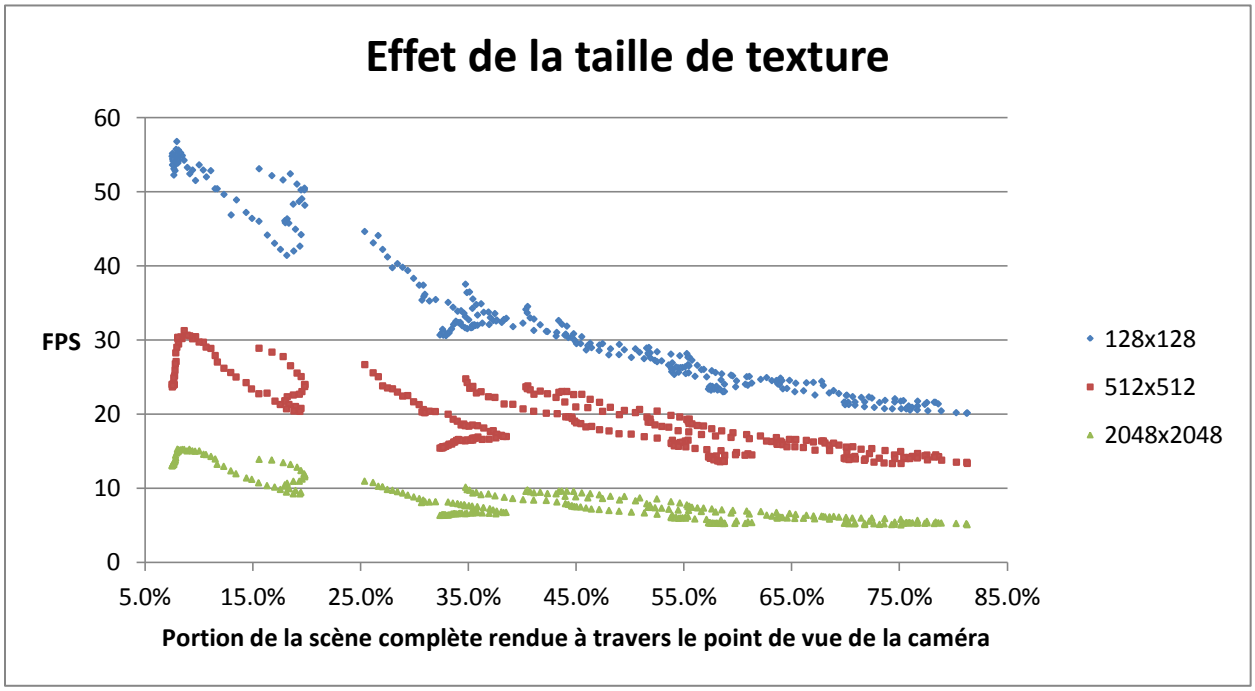

Figure 62 : Effet de la taille de la texture d'une tuile selon le pourcentage de la scène rendue

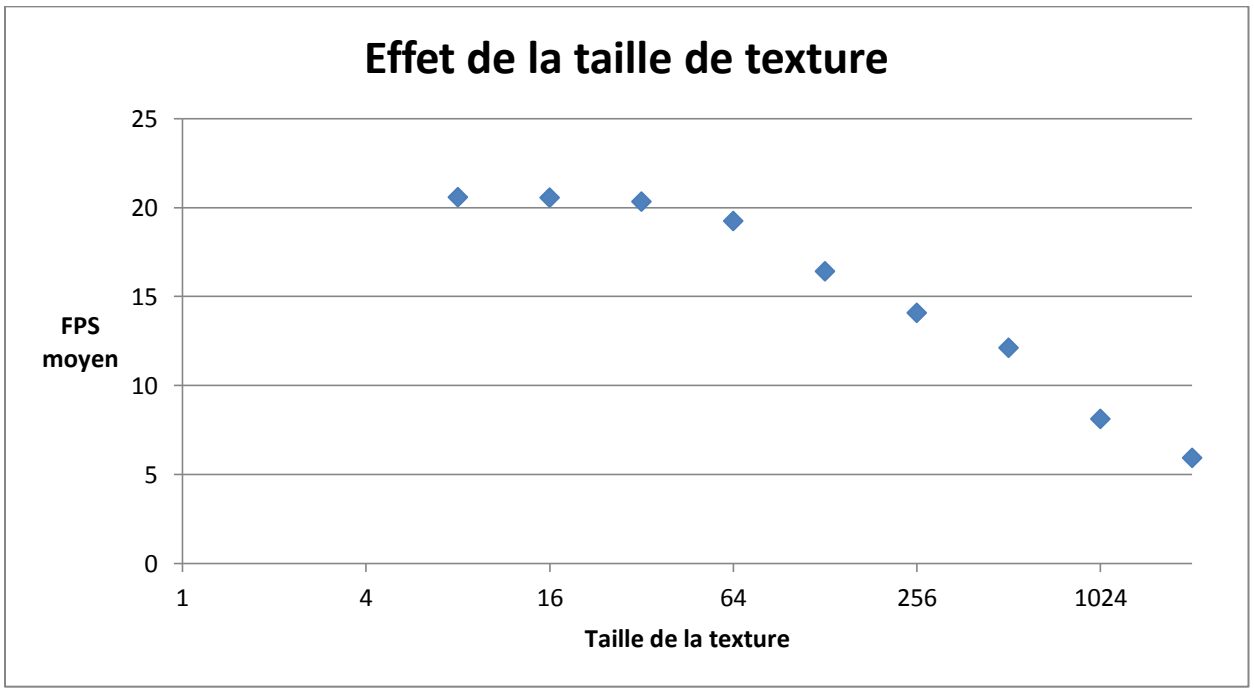

Figure 63 : Effet de la taille de la texture d'une tuile

Test sur la profondeur des tuiles à haute résolution

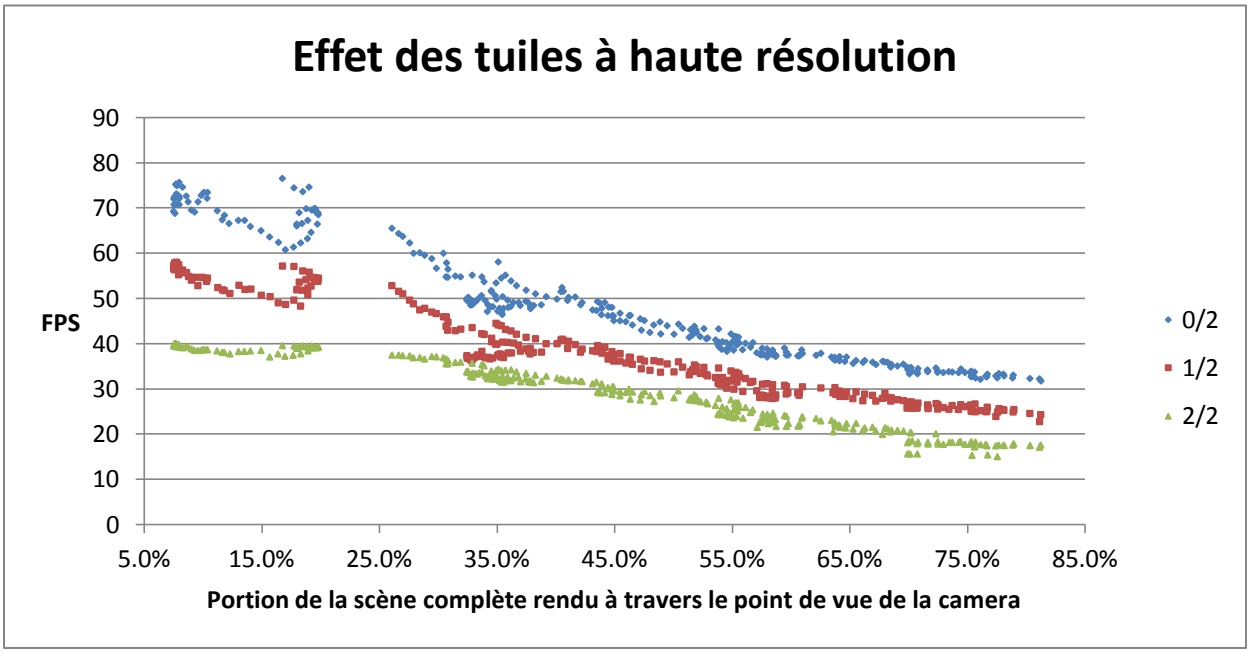

Figure 64 : Effet des tuiles à haute résolution selon le pourcentage de la scène rendue

Test sur l'effet du brouillard

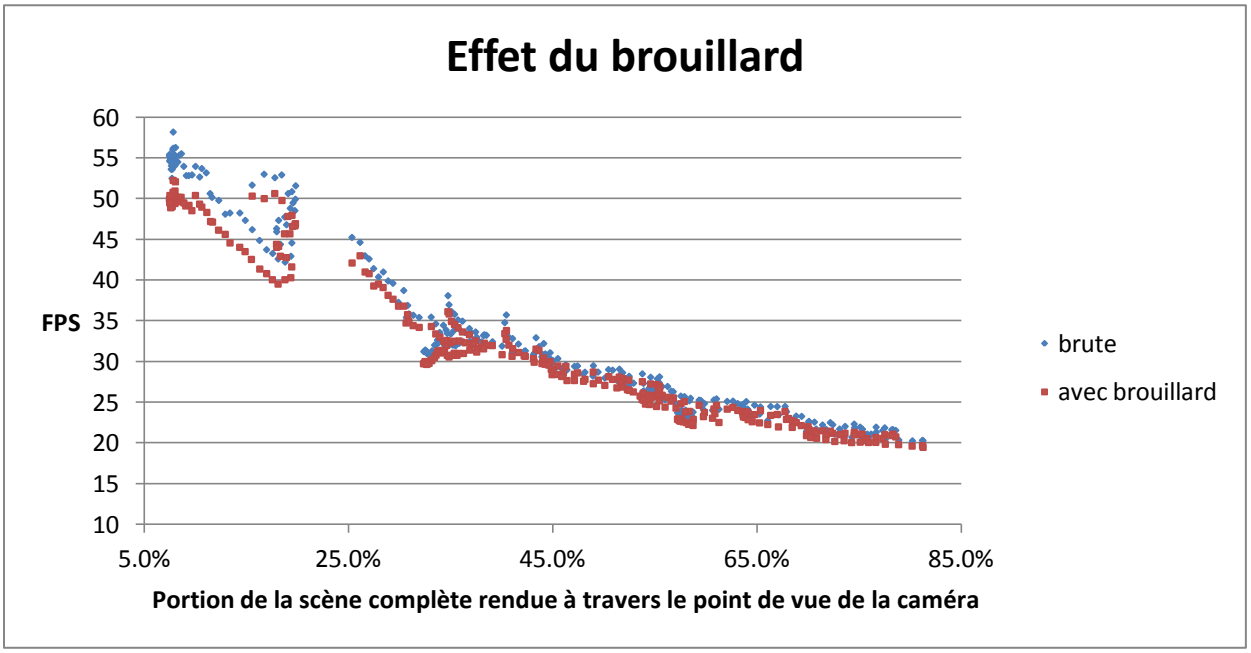

Figure 65 : Effet du brouillard selon le pourcentage de la scène rendue

Test sur l'effet d'anticrénelage

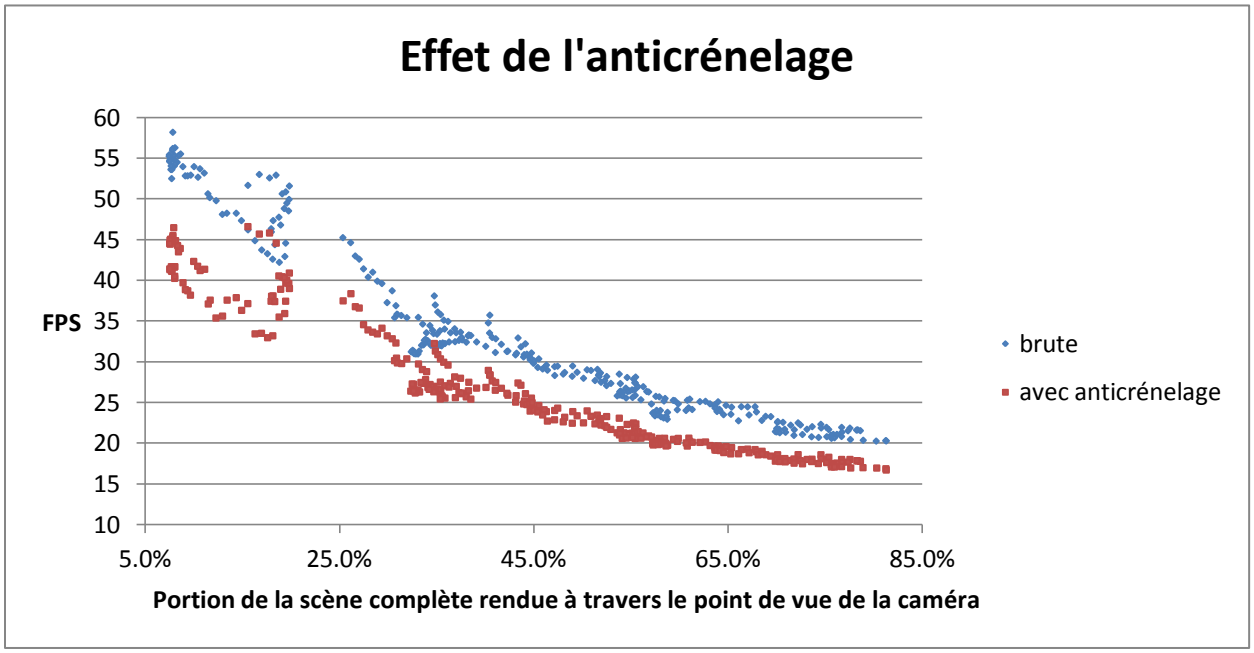

Figure 66 : Effet de l'anticrénelage selon le pourcentage de la scène rendue

Résultats sur l'ordinateur de bureau

Tests sur la résolution des tuiles de terrain :

Le premier test fait varier le nombre de sommets qui définissent une tuile de terrain. Trois valeurs ont été choisies afin d'avoir des tuiles de 15x15, 31x31 et 45x45 sommets par tuile. Le choix de ces valeurs provient de l'application SVS étudiée. Nous nous sommes basés sur les paramètres actuels de l'application pour des fins de référence et avons choisi des valeurs de tests qui varient près de cette valeur. L'application utilise des tuiles constituées de 25x25 sommets. Tous les autres paramètres sont fixés à :

- grille d'une dimension de 25x25 tuiles,
- texture placée sur les tuiles de taille 128x128 pixels RGBA,
- une taille de la fenêtre de projection de 640x480 pixels,
- l'intégralité des tuiles sont à haute résolution (tous les sommets sont utilisés pour définir la scène 3D).

Le graphique illustré à la Figure 67 montre l'évolution du FPS par image selon la portion de la scène rendue à l'écran pour les trois valeurs de sommets par tuile.

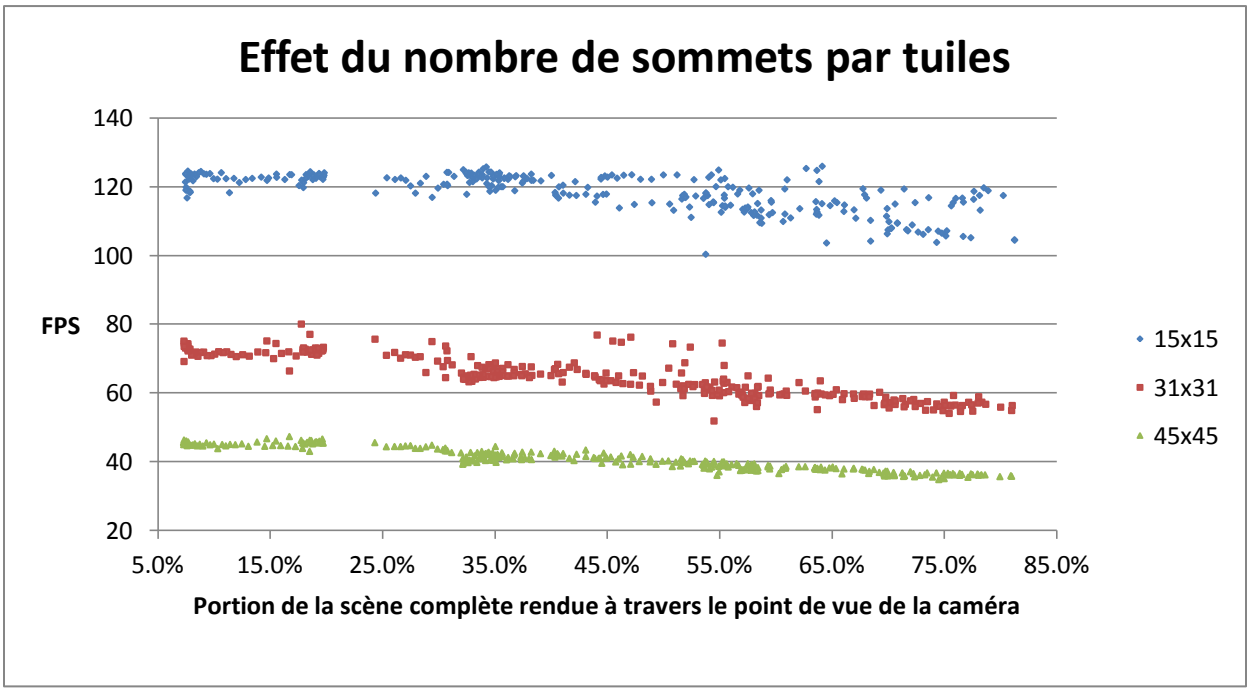

Figure 67 : Effet du nombre de sommets par tuiles selon le pourcentage de la scène rendue

Un deuxième test a été réalisé cette fois-ci afin de mettre l'accent sur la dégradation des performances due à l'augmentation du nombre de sommet par tuile. Le graphique illustré à la Figure 68 montre l'évolution du FPS moyen selon la quantité de polygone qui constitue la scène 3D. Les valeurs testées pour ce test sont les suivantes : 9x9, 13x13, 17x17, 21x21, 25x25, 31x31, 37x37, 45x45 et 53x53 sommets par tuile. Ces valeurs ont été traduites en nombre de polygones constituant la scène pour l'évolution du FPS moyen selon le niveau de détail de la scène. Tous les autres paramètres sont fixes. Chaque échantillon correspond au FPS moyen établi sur l'ensemble des temps de traitement par image d'un test contrairement au test précédent qui calculait le temps de traitement par image.

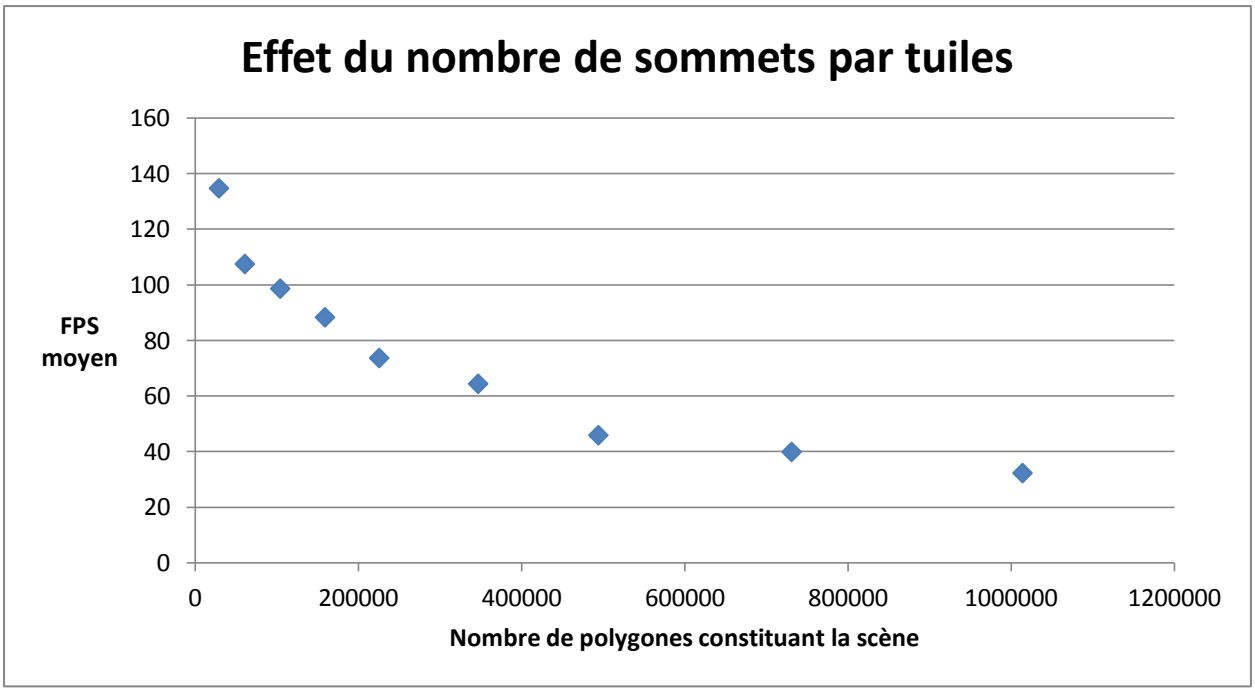

Figure 68 : Effet du nombre de sommets par tuiles

Tests sur la taille de la grille de tuiles

Le premier test effectué fait varier la taille de la grille de tuiles qui représente la scène. Trois valeurs ont été choisies de sorte que nous ayons des grilles de terrain de dimension : 15x15, 31x31 et 45x45 tuiles. Ces valeurs sont les mêmes que celles du premier test portant sur l'effet du nombre de sommets par tuile pour avoir le même nombre de sommets qui définissent la scène 3D dans les deux séries de tests. Les autres paramètres sont fixés à :

- la résolution d'une tuile est de 25x25 sommets,
- la texture placée sur les tuiles à une taille de 128x128 pixels RGBA,
- la taille de la fenêtre de projection est de 640x480 pixels,
- l'intégralité des tuiles sont à haute résolution (tous les sommets sont utilisés pour définir la scène 3D).

Le graphique illustré à la Figure 69 montre l'évolution du temps de traitement par image (en FPS) selon la portion de la scène rendue à l'écran pour les trois valeurs du test.

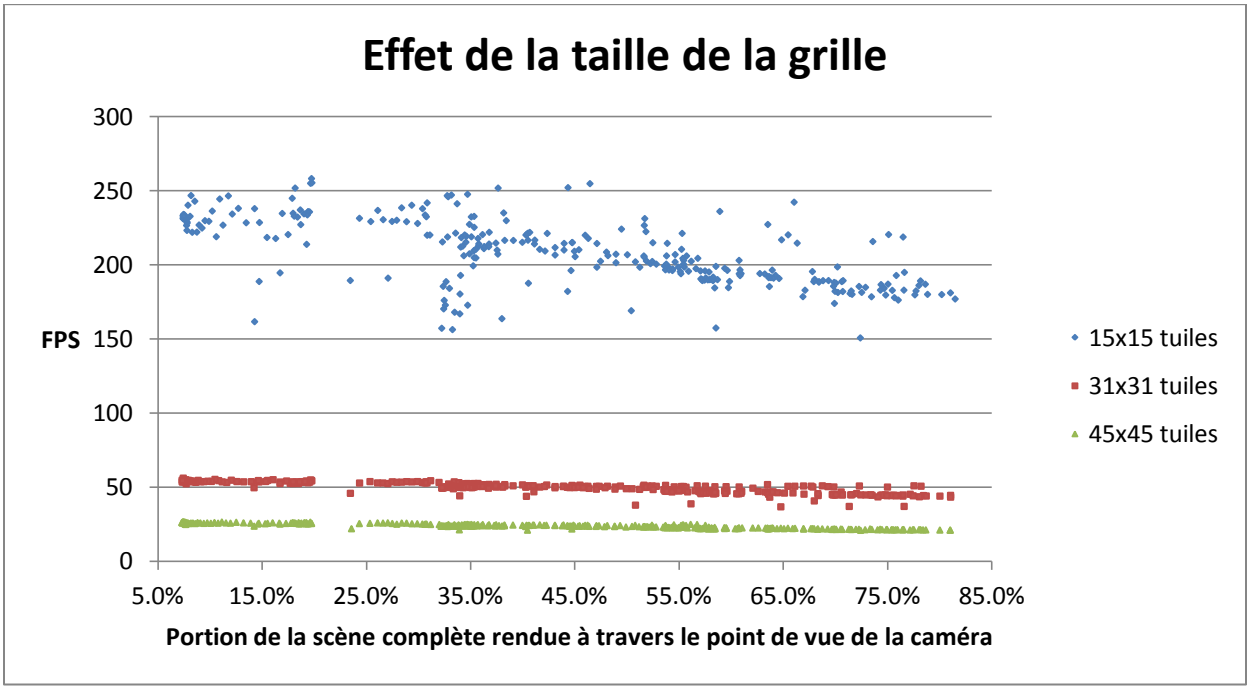

Figure 69 : Effet de la taille de la grille de tuiles selon le pourcentage de la scène rendue

Un deuxième test a été réalisé afin de mettre l'accent sur la dégradation des performances due à l'augmentation du nombre de tuiles de terrain composant la scène. Le graphique illustré à la Figure 70 montre l'évolution du FPS moyen selon la quantité de polygone qui constitue la scène 3D. Les valeurs testées pour ce test sont les suivantes : 9x9, 13x13, 17x17, 21x21, 25x25, 31x31, 37x37, 45x45 et 53x53 tuiles. Encore une fois, tous les autres paramètres sont fixes et chaque échantillon correspond au FPS moyen établi sur l'ensemble des temps de traitement par image d'un test.

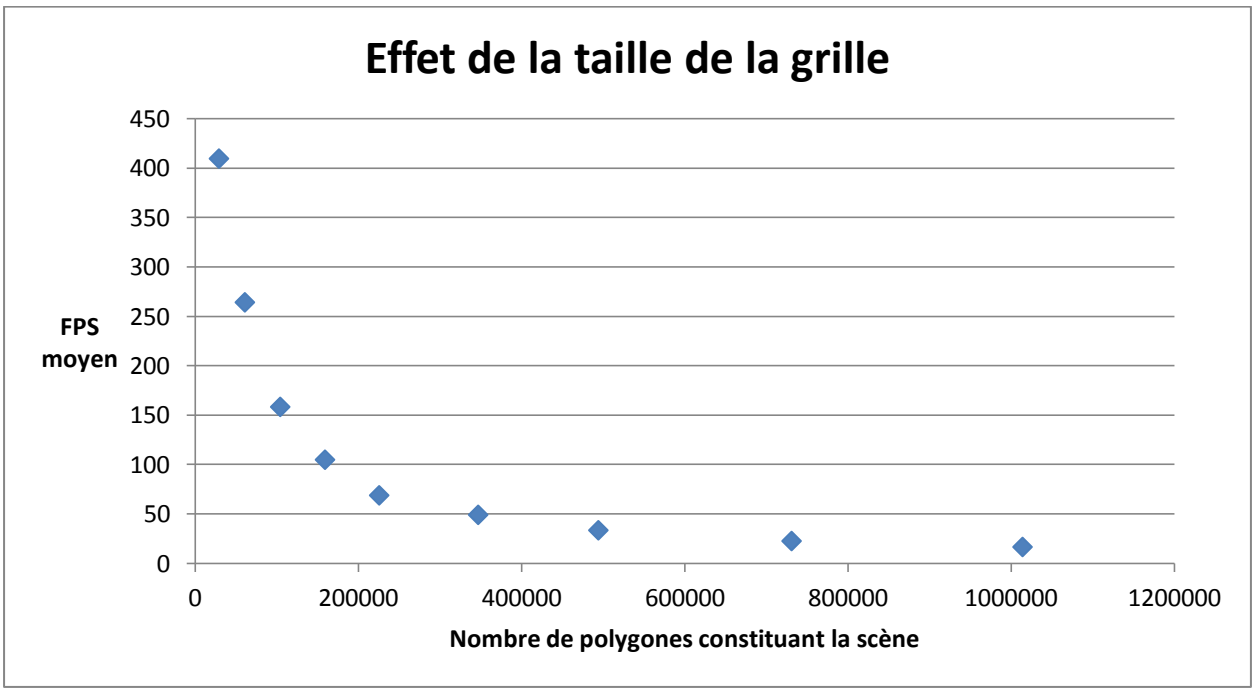

Figure 70 : Effet de la taille de la grille de tuiles

Tests sur la résolution de la fenêtre de projection

Ce test fait varier la résolution de la fenêtre de projection. Trois valeurs ont été choisies afin d'obtenir des fenêtres avec les résolutions suivantes : 640x480, 1024x768 et 1280x960 pixels, tous des formats 4:3. Notons que ce format représente le format usuel dans l'industrie aéronautique et que la résolution 640x480 est considérée comme la résolution « moderne ». Tous les autres paramètres sont fixes pour la réalisation de ce test :

- la grille de tuiles à une dimension de 25x25 tuiles,
- les tuiles de terrain sont composées de 25x25 sommets,
- la texture placée sur les tuiles à une taille de 128x128 pixels RGBA,
- l'intégralité des tuiles sont à haute résolution.

Le graphique illustré à la Figure 71 montre l'évolution du FPS par image selon la portion de la scène rendue à l'écran pour les trois valeurs du test.

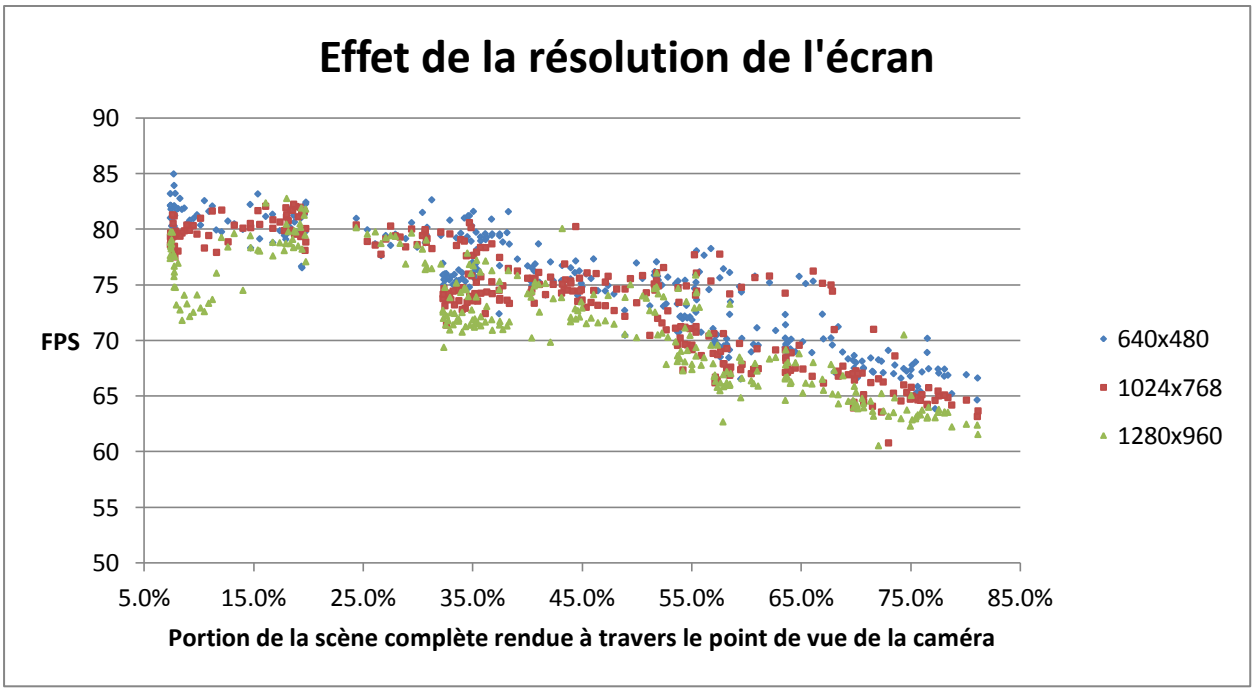

Figure 71 : Effet de la résolution de la fenêtre d'affichage selon le pourcentage de la scène rendue

Un deuxième test a été réalisé afin de bien visualiser l'impact sur les performances obtenues par la variation de la résolution de la fenêtre d'affichage. Le graphique illustré à la Figure 72 montre l'évolution du FPS moyen selon la résolution de la fenêtre d'affichage. Les valeurs utilisées pour ce test sont : 640x480, 800x600, 1024x768, 1152x864 et 1280x960. Cette plage de résolutions couvre toutes les résolutions en format 4:3 standards qu'il était possible d'afficher sur la plateforme cible. Tous les autres paramètres sont fixes et chaque échantillon correspond au FPS moyen établi sur l'ensemble des temps de traitement par image d'un test.

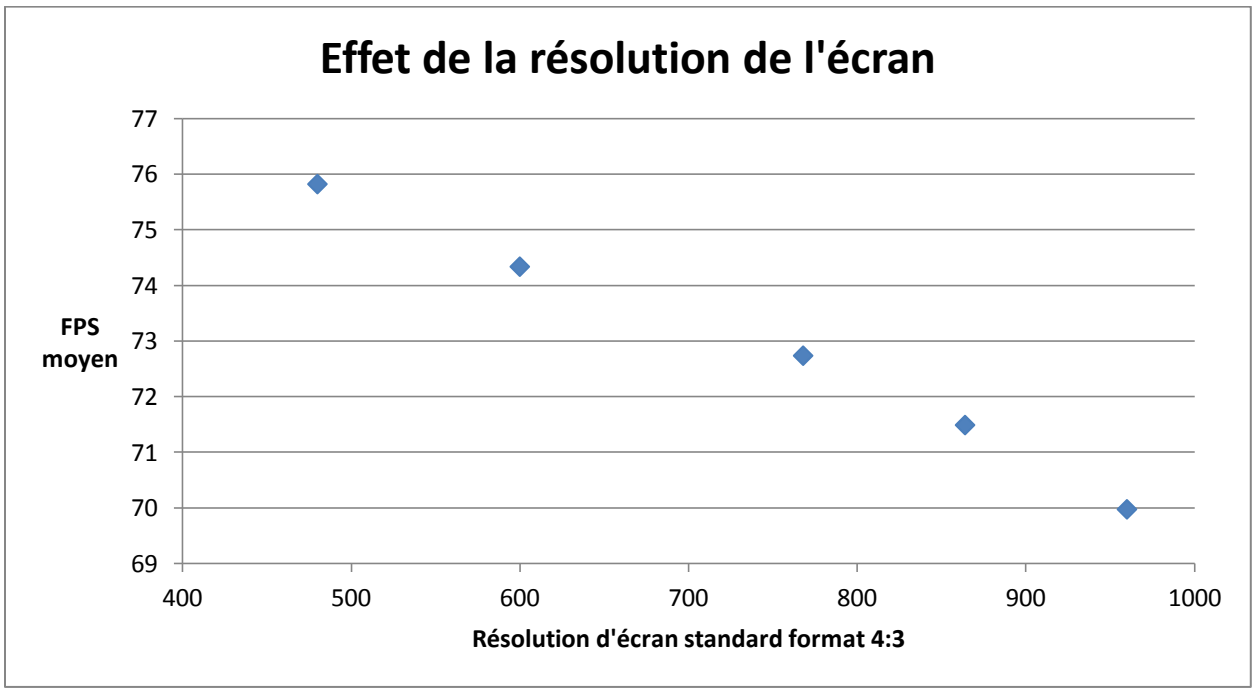

Figure 72 : Effet de la résolution de la fenêtre d'affichage

Test sur la taille de la texture des tuiles

Ce test fait varier la taille de la texture qui couvre les tuiles de terrain. Notons qu'il s'agit de la même texture (ressource) qui est appliquée sur chaque tuile. Les trois valeurs retenues pour ce test sont des textures de dimension : 128x128, 512x512 et 2048x2048 pixels. Le choix de ces valeurs provient de l'application SVS étudiée. Nous nous sommes basés sur les paramètres actuels de l'application pour des fins de référence et avons choisi des valeurs de tests qui varient près de cette valeur. Les autres paramètres resteront fixes :

- la grille de tuiles à une dimension de 25x25 tuiles,
- la résolution d'une tuile de terrain est de 25x25 sommets,
- la taille de la fenêtre de projection est de 640x480 pixels,
- l'intégralité des tuiles sont à haute résolution.

Le graphique illustré à la Figure 73 montre l'évolution du FPS par image selon la portion de la scène rendue à l'écran pour les trois valeurs du test.

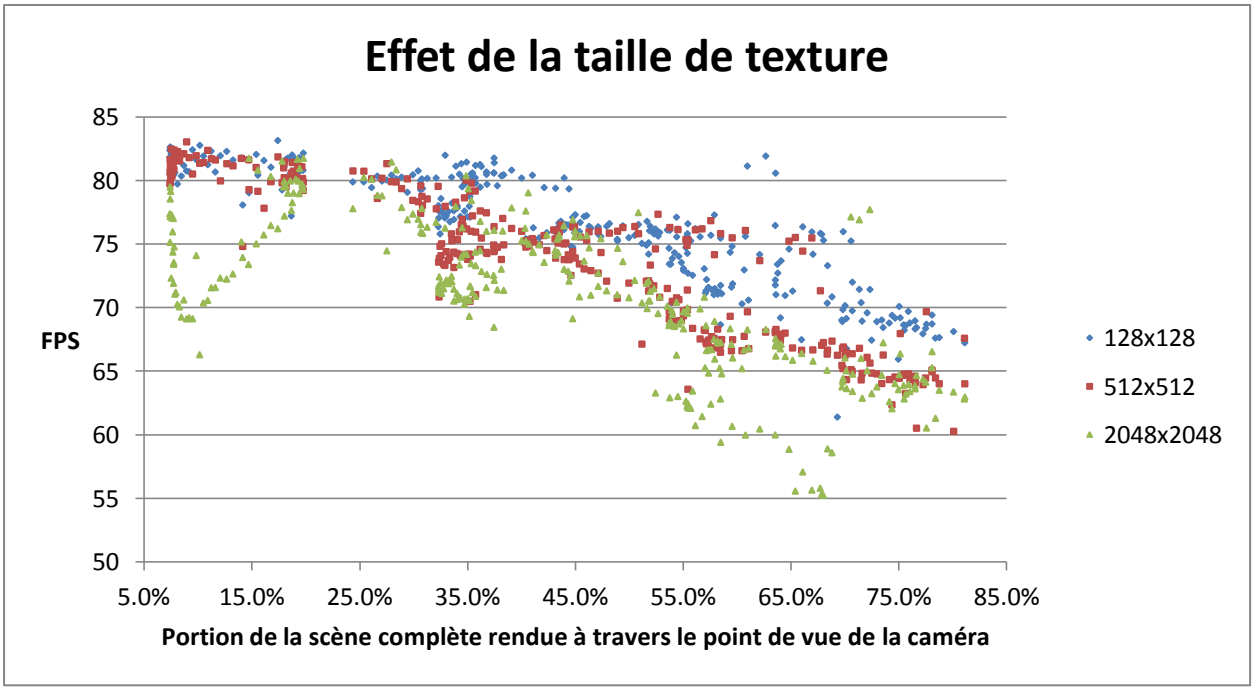

Figure 73 : Effet de la taille de la texture d'une tuile selon le pourcentage de la scène rendue

Un deuxième test a été réalisé afin de bien visualiser l'impact sur les performances obtenues par la variation de la taille de la texture placée sur les tuiles de terrain. Le graphique illustré à la Figure 74 montre l'évolution du FPS moyen selon la taille de texture utilisée pour le test. Les valeurs utilisées sont : 8x8, 16x16, 32x32, 64x64, 128x128, 256x256, 512x512, 1024x1024, 2048x2048 et 4096x4096 pixels. Tous les autres paramètres sont fixes et chaque échantillon correspond au FPS moyen établi sur l'ensemble d'un test.

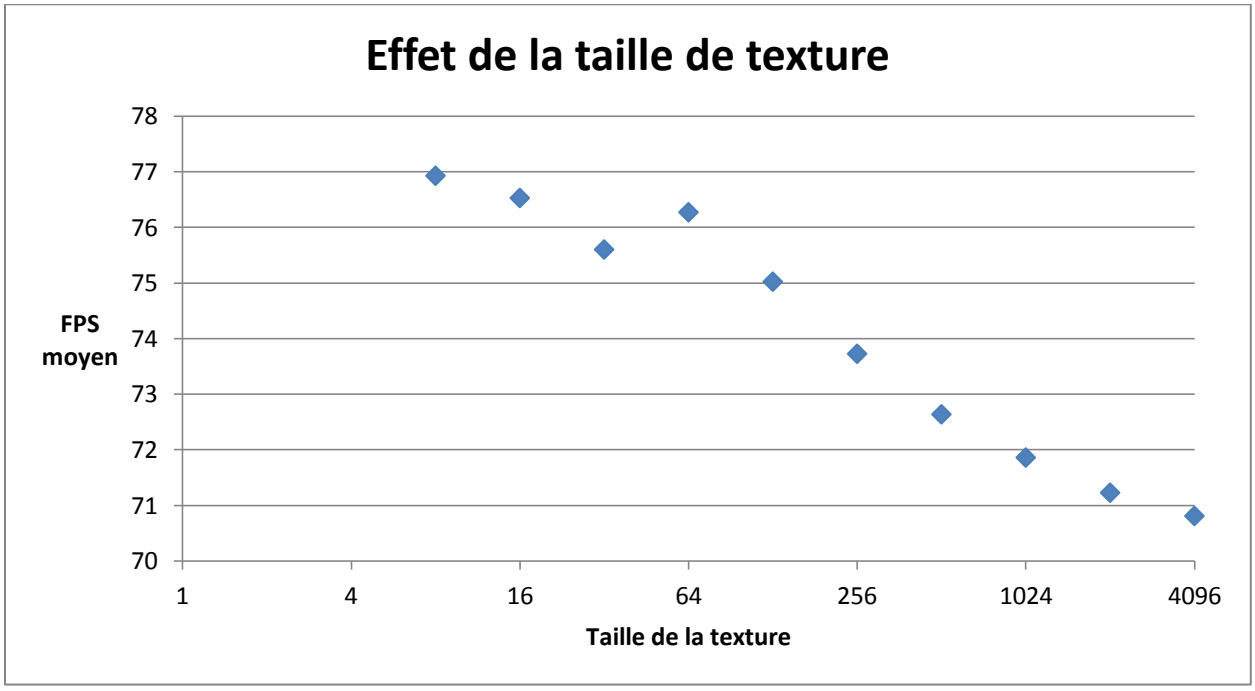

Figure 74 : Effet de la taille de la texture d'une tuile

Test sur la profondeur des tuiles à haute résolution

Ce test fait varier la profondeur des tuiles à haute résolution par rapport à la tuile centrale. La tuile centrale est toujours à haute résolution et plus la « profondeur » est grande, plus grand sera le ratio entre le nombre de tuiles de haute résolution et celles, limitrophes, de basse résolution. Trois ratios de profondeur ont été choisis afin de réaliser ce test : 0/2, 1/2 et 2/2. Pour la réalisation du test avec un ratio de profondeur de tuiles à haute résolution de 1/2 (donc avec la moitié (1/2) des tuiles situées entre la tuile centrale et la limite du terrain à haute résolution et les subséquentes à basse résolution) et en se référant au schéma de la Figure 75, seules les tuiles vertes et bleues seront à haute résolution. Par conséquent, un ratio de 0/2 implique que seule la tuile bleue est à haute résolutions et qu'un ratio de 2/2 implique que la totalité des tuiles sont à haute résolution. Tous les autres paramètres sont fixes pour la réalisation de ce test :

- la grille de tuiles à une dimension de 25x25 tuiles,
- la résolution d'une tuile de terrain est de 25x25 sommets,
- la texture placée sur les tuiles à une taille de 128x128 pixels RGBA,
- la taille de la fenêtre de projection est de 640x480 pixels.

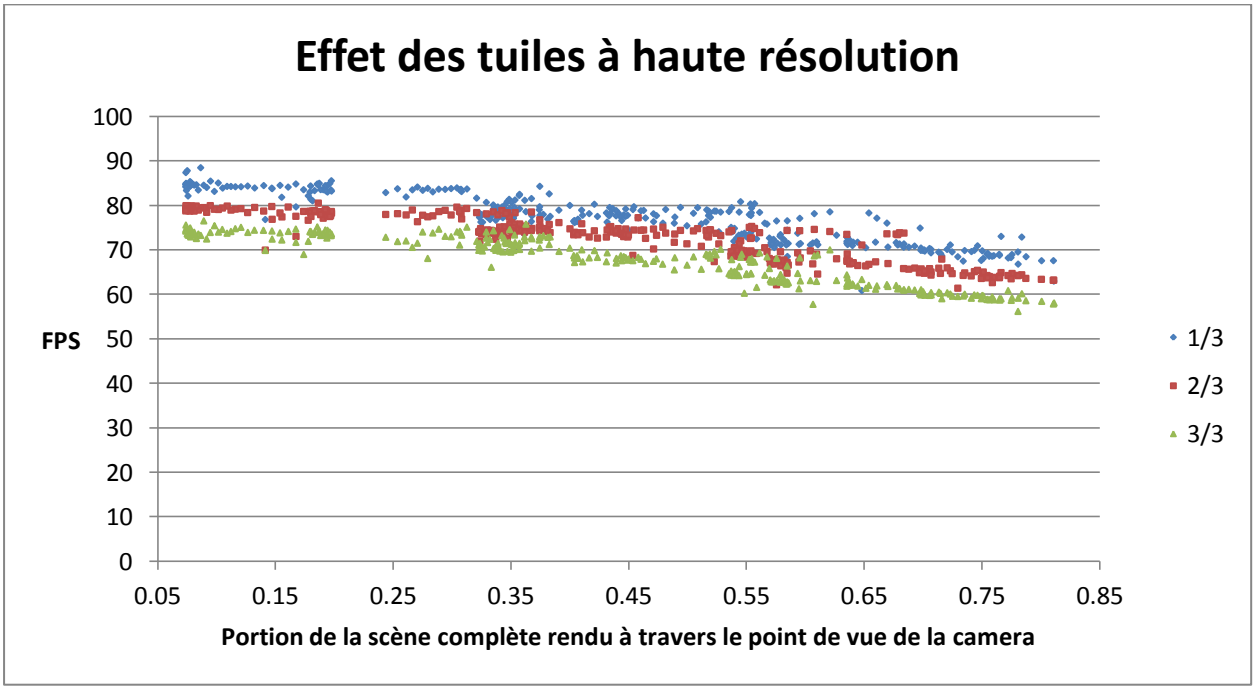

Figure 75 : Effet des tuiles à haute résolution selon le pourcentage de la scène rendue

Test sur l'effet du brouillard

Ce test mesure l'impact de l'ajout d'un effet de brouillard sur la scène. Pour effectuer ce test nous avons utilisé une grille de tuiles de dimension 31x31, une résolution de tuile de terrain de 31x31 sommets, une texture de tuile ayant une taille de 128x128 pixels RGBA, une taille de fenêtre de projection de 640x480 pixels et notons que l'intégralité des tuiles sont à haute résolution.

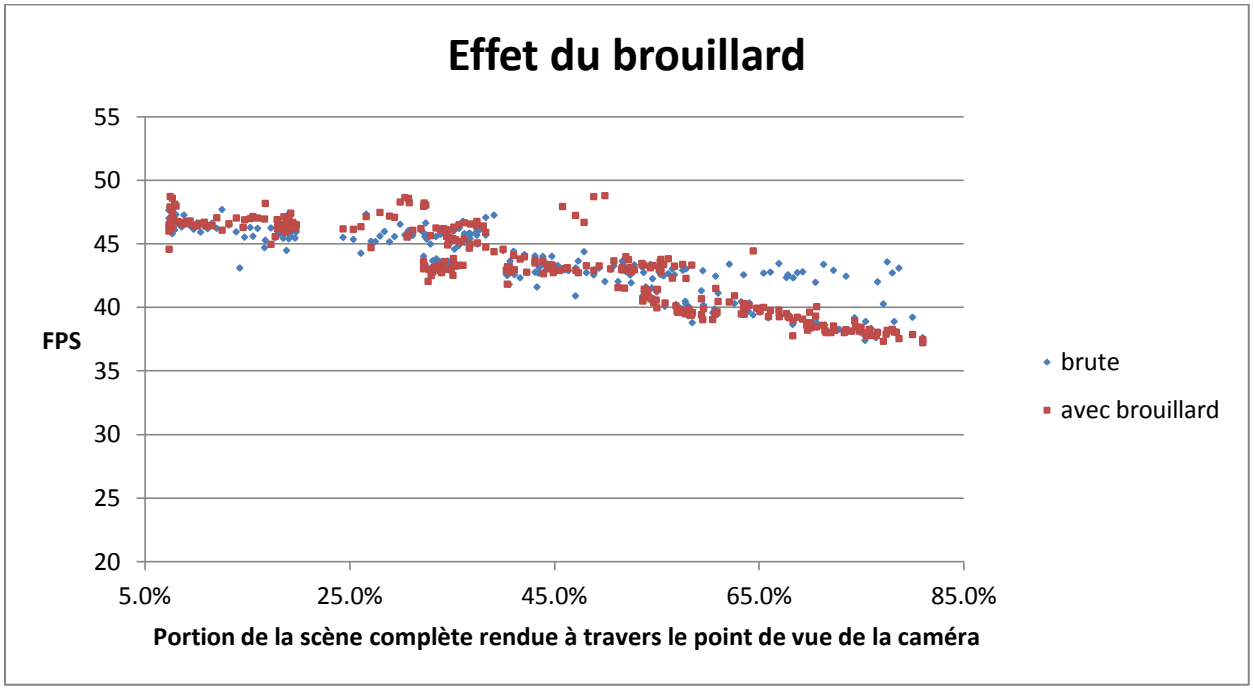

Figure 76 : Effet du brouillard selon le pourcentage de la scène rendue

Test sur l'effet d'anticrénelage

Ce test mesure l'impact de l'ajout d'un mécanisme d'anticrénelage par rapport au rendu final de la scène. Pour effectuer ce test nous avons utilisé une grille de tuiles de dimension 31x31, une résolution de tuile de terrain de 31x31 sommets, une texture de tuile ayant une taille de 128x128 pixels RGBA, une taille de fenêtre de projection de 640x480 pixels et notons que l'intégralité des tuiles sont à haute résolution.

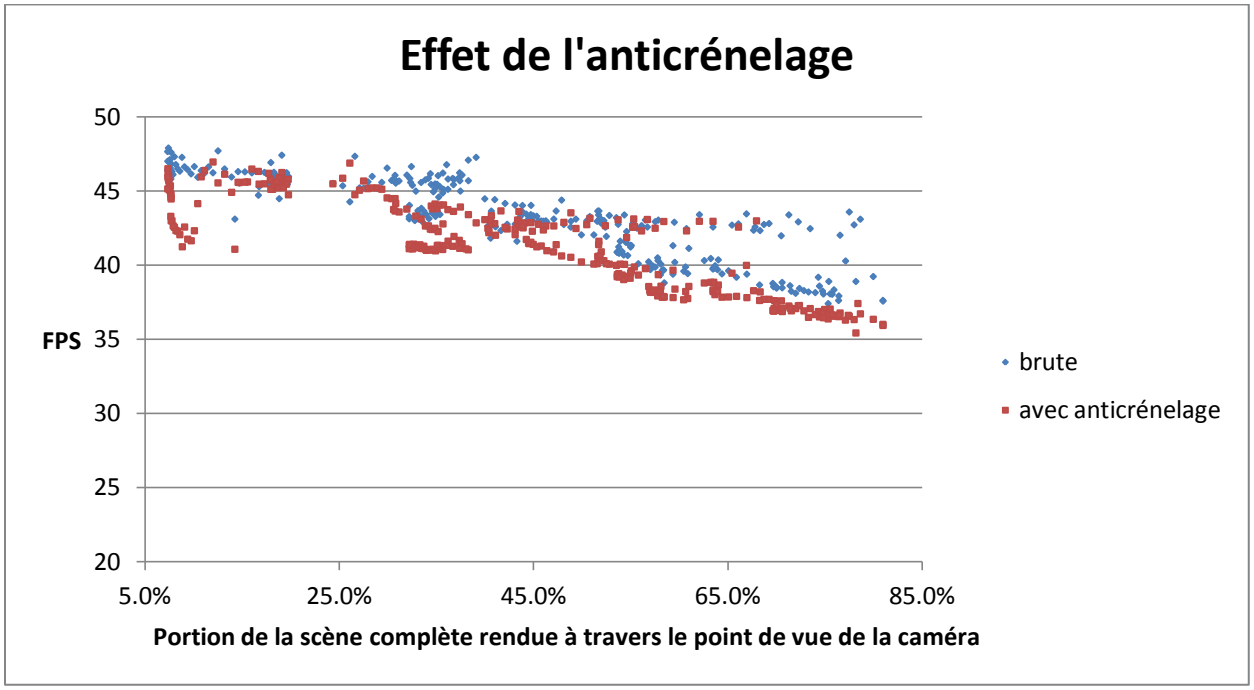

Figure 77 : Effet de l'anticrénelage selon le pourcentage de la scène rendue

ANNEXE B : DÉTAILS TECHNIQUES DE L'OUTIL DÉVELOPPÉ

Lecteur de tests

Le lecteur de tests a pour utilité d'aller lire tous les tests inscrits dans le fichier de déclaration des tests nommé *tests-setup.txt*. C'est dans ce fichier, qui est situé dans le répertoire racine du programme, qu'un utilisateur entre tous les tests qu'il souhaite exécuter. Le lecteur de tests lira chacun de ces tests définis sous format texte et les transformera un à un en objet structure de test (*STest*). Les objets *STest* sont alors stockés dans un tableau qui sera ultérieurement passé au prochain bloc de traitement qui les exécutera. Un objet structure de test *STest* comprend les champs suivants :

1. *testType* : Type de test à exécuter.
2. *tileResolution* : Nombre de sommets par tuile de terrain.
3. *gridSize* : Nombre de tuiles de terrain composant la scène.
4. *textureSize* : Taille de la texture utilisée.
5. *screenHeight* : Hauteur de la fenêtre abritant le contexte OpenGL.
6. *hiresdepth* : Profondeur, en nombre de tuiles, des tuiles à haute résolution. Cette profondeur est calculée à partir de la tuile centrale de la scène.
7. *camheigth* : Hauteur à laquelle la caméra commence l'exécution des tests. Cette valeur est exprimée en pourcentage de la hauteur maximale que la caméra peut atteindre.
8. *values* : Tableau de valeurs à tester. Ces valeurs sont relatives au type de test à exécuter.
9. *nbvalues* : Nombre de valeurs dans le tableau de valeurs à tester.

Expliquons maintenant comment les tests sont déclarés par l'utilisateur et ce que représentent plus concrètement les champs de la structure *STest*.

Déclaration d'un test

L'utilisateur spécifie un test sous forme de commande. Pour déclarer un test, il doit indiquer le type de test qu'il souhaite exécuter et ajouter les attributs propres à ce dernier. Voici un exemple de commande de test à inscrire dans le fichier *tests-setup.txt* :

```
tileresolution -gridsize 39 -texsize 256 -scrsize 720 -values 25 35 45 55
```

Le premier mot-clé de cette commande (tileresolution) est le type de test à exécuter (un test de résolution des tuiles dans le cas présent). Le deuxième mot-clé (-gridsize) est un attribut qui spécifie la dimension de la scène. Notons que tous les attributs sont précédés d'un signe tiret (-) et que l'ordre dans lequel ils sont spécifiés n'a pas d'importance. Le troisième mot-clé (-texsize) indique la taille de la texture, en texels, à utiliser pour le test. Le quatrième mot-clé (-scrsize) précise la hauteur de la fenêtre dans laquelle seront exécutés les tests graphiques. Le dernier attribut (-values) sert à spécifier les valeurs pour lesquelles on souhaite effectuer le test. Cet exemple de commande sert donc à lancer un test de résolution des tuiles qui générera 4 scènes qui auront toutes une dimension de 39 tuiles par 39 tuiles, une texture de 256 par 256 texels et une fenêtre de rendu ayant une hauteur de 720 pixels en respectant un ratio de 4:3. Cependant, les scènes de ces tests auront des résolutions de tuiles différentes : 25X25, 35X35, 45X45 et 55X55 sommets. Une description détaillée pour chacun des mots-clés suivra dans les prochaines sous-sections.

Tests disponibles

Quatre types de test sont disponibles. Ceux-ci sont répertoriés dans le Tableau 8. Les noms de ces derniers sont basés sur le paramètre qui variera lors de l'exécution du test en question.

Tableau 8 : Types de test disponibles et définitions

Nom des tests	Description
Résolution des tuiles ***tileresolution***	Ce test va générer une série de scènes (terrains) avec un nombre de sommets par tuile de terrain variable. Les dimensions d'une tuile resteront fixes, mais le nombre de sommets qui les constituent augmentera ou diminuera dans le cas échéant. La résolution d'une tuile doit être un nombre impair.

Grandeur de la scène ***gridsize***	Ce test va générer une série de scènes constituée d'un nombre de tuiles variable. Les caractéristiques des tuiles resteront inchangées, mais leur nombre augmentera ou diminuera dans le cas échéant. La dimension (en nombre de tuiles) de la scène doit être un nombre impair.
Taille de la texture ***texturesize***	Ce test va générer une série de scènes avec diverses tailles de texture. Pour chacun des tests, les propriétés du terrain resteront fixes sauf pour les tailles de textures qui varieront selon les valeurs demandées par l'utilisateur. En tout temps, une seule texture est chargée en mémoire. Celle-ci est ensuite appliquée sur chacune des tuiles de terrain de la scène. La taille d'une texture doit être une puissance de 2.
Résolution de l'écran ***screensize***	Ce test va générer une série de fenêtres aux dimensions variables dans lesquelles sera affichée la scène tridimensionnelle. Toutes les propriétés graphiques seront gardées de test en test, seules les dimensions de la fenêtre changeront. Les fenêtres générées respecteront toujours le format 4:3. Pour déclarer ce type de test, l'utilisateur n'aura qu'à fournir la dimension verticale (en pixels) de la fenêtre. La dimension horizontale sera automatiquement calculée pour respecter le format 4:3.

Attributs pour les tests

Sept attributs sont disponibles pour exécuter les tests. Ceux-ci sont énumérés dans le Tableau 9. Encore une fois, leurs noms sont basés sur la propriété graphique qu'ils affecteront. Selon le test à exécuter, certains de ces attributs sont obligatoires.

Tableau 9 : Types d'attribut disponibles et définitions

Nom des attributs	Description
Résolution des tuiles ***-tileres*** (obligatoire)	Cet attribut définit le nombre de sommets par tuile de terrain. La valeur *n* associée à cet attribut indique que les tuiles seront composées de *n* par *n* sommets. Cet attribut est obligatoire pour tous les tests excepté pour le test de résolution des tuiles. La résolution d'une tuile doit être un nombre impair.
Grandeur de la scène ***-gridsize***	Cet attribut spécifie le nombre de tuiles qui compose une scène tridimensionnelle. La valeur *n* associée à cet attribut indique que la dimension de la scène sera de *n* par *n* tuiles. Cet attribut est obligatoire

(obligatoire)	pour tous les tests excepté pour le test de grandeur de la scène. Les dimensions de la scène doivent être un nombre impair.
Taille de la texture ***-texsize*** (obligatoire)	Cet attribut indique la taille de la texture qui recouvrera les tuiles de la scène. La valeur *n* associée à cet attribut indique que la taille de la texture utilisée sera de *n* par *n* texels. Cet attribut est obligatoire pour tous les tests excepté pour le test de taille de la texture. La taille d'une texture doit être une puissance de 2. Une valeur plus petite ou égale à zéro signifie que le test s'effectuera sans texture.
Résolution de l'écran ***-scrsize*** (obligatoire)	Cet attribut précise la dimension de la fenêtre dans laquelle sera affichée la scène tridimensionnelle. La valeur *n* associée à cet attribut indique que la dimension de la fenêtre de projection sera d'une hauteur de *n* pixels. La dimension de la largeur de la fenêtre sera automatiquement calculée par l'outil afin de respecter un ratio de 4:3. Cet attribut est obligatoire pour tous les tests excepté pour le test de résolution de l'écran.
Profondeur des tuiles à haute résolution ***-hires***	Cet attribut spécifie la profondeur des tuiles à haute résolution par rapport à la tuile centrale. Il existe deux types de résolution de tuile : haute et basse. Sur une tuile à haute résolution, tous les sommets de la tuile sont rendus à l'écran tandis que sur une tuile à basse résolution, seulement un sommet sur deux est rendu à l'écran. La valeur minimale et par défaut de cet attribut est 1 ce qui signifie que la tuile centrale est toujours de haute résolution. Lorsqu'on augmente cette valeur, les tuiles autour de la tuile centrale seront-elles aussi affichées à haute résolution. La valeur maximale que peut prendre cet attribut est égale à : (grandeur de la scène + 1) / 2. En d'autres termes, cette valeur signifie que toutes les tuiles de la scène sont affichées à haute résolution.
Hauteur de la caméra ***-maxheight***	Cet attribut précise la hauteur à laquelle la caméra commence l'exécution des tests. Cette valeur est exprimée en pourcentage de la hauteur maximale que la caméra peut atteindre. Cette hauteur est prédéterminée par l'outil et correspond à la hauteur à laquelle la scène devient plus petite que la fenêtre d'affichage. La valeur maximale et par défaut de cet attribut est 100 et la valeur minimale est 25.
Valeurs à tester ***-values***	Cet attribut annonce l'énumération des valeurs à tester par le test principal. Les valeurs sont énumérées les unes après les autres et sont séparées par un espace. Notons que pour chacune des valeurs une courbe sera générée, et ce, sur le même graphique.

Effet de brouillard ***-fog***	Cet attribut ajoute la fonctionnalité graphique de brouillard dynamique à la scène.
Anticrénelage ***-antialiasing***	Cet attribut déclenche la fonctionnalité d'anticrénelage lors de la génération des images de synthèse.

Vérifications des paramètres

Afin de rendre notre outil robuste, nous avons implémenté un système de vérification pour le module lecteur de test. Plusieurs validations et rectifications sont effectuées lors de la lecture des tests. Voici la liste des éléments qui sont affectés par ces vérifications :

1. La résolution des tuiles et la grandeur de la scène doivent être de nombre impair pour les raisons expliquées à la section 3.2. Si l'utilisateur entre par mégarde une valeur paire, celle-ci sera augmentée de 1 unité afin de respecter la contrainte. Les valeurs modifiées par l'outil apparaitront dans le fichier des métriques de sortie (fichier log). Une valeur plus petite que 3 pour la grandeur de la scène ou une valeur plus petite que 5 pour la résolution des tuiles invalide le test.
2. La taille des textures (en texels) doit être une puissance de 2. Il en est ainsi, car OpenGL SC ne supporte que ce type de format. Si l'utilisateur rentre une autre valeur qui ne respecte pas cette condition alors, la valeur sera incrémentée à la prochaine puissance de 2. La valeur retenue sera mentionnée dans le fichier des métriques de sortie (fichier log). D'autre part, si l'utilisateur entre une valeur plus petite ou égale à 0 alors, le test s'effectuera sans texture.
3. Si la valeur de profondeur des tuiles à haute résolution (-hires) ou celle de la hauteur de caméra (-maxheight) se trouve hors de leurs plages de valeurs permises respectives, la valeur de l'attribut sera remplacée par l'extrémum permis le plus prêt.
4. Le lecteur de test supprime les valeurs doublons de la liste des valeurs à tester pour éviter de faire le même test plusieurs fois.

5. Une commande de test doit toujours commencer par un nom de test valide. Si cette contrainte n'est pas respectée, le test sera tout simplement ignoré et le lecteur passera au test suivant.
6. Si un attribut invalide est décelé par le lecteur (ex : -toto), ce dernier invalide le test et lecteur passe au test suivant.
7. Si l'utilisateur inscrit plus d'une valeur à un attribut qui ne peut en contenir qu'un seul, seule la première valeur sera retenue et les autres seront ignorées. Par conséquent, ceci n'invalide pas un test. La valeur retenue sera mentionnée dans le fichier des métriques de sortie (fichier log).
8. L'ordre des attributs n'a pas d'importance.
9. Le nombre de retour de chariot entre les commandes de test et le nombre d'espaces entre les mots-clés, ainsi que leurs valeurs connexes, n'ont pas d'importance.

Une fois tous les tests lus, ceux qui auront passé avec succès l'étape de validation seront envoyés au prochain module afin d'entamer leur exécution.

Génération des objets 3D de forme pyramidale

Voici la description du code qui génère les sommets des tuiles de terrain de notre banc de test :

```
void CTile::GenerateVertices(int pNoize)
{
   mVertices = new CVec3f[mHiResVerticesCount];

   srand ((unsigned int)time(NULL));

   for(unsigned int i = 0; i < mHiResVerticesPerSide; ++i)
   {
      for(unsigned int j = 0; j < mHiResVerticesPerSide; ++j)
      {
         for(unsigned int k = 0; k < 3; ++k)
         {
            if(k == 0) // X
            {
               mVertices[j + i * mHiResVerticesPerSide].Set_X((GLfloat)j  /
               (GLfloat)(mHiResVerticesPerSide - 1) * (GLfloat)mTileSize);
            }
            else if(k == 1) // Y
            {
```

```
// Lower left corner

// vertex (0, 0) AND with a left neighbour
if(i == 0 && j == 0 && mLeftTile != NULL)
{
   mVertices[j + i * mHiResVerticesPerSide].Set_Y(
      mLeftTile->GetVertex(0, mHiResVerticesPerSide - 1).Get_Y()
      );
}
// vertex (0, 0) AND with a down neighbour
else if(i == 0 && j == 0 && mDownTile != NULL)
{
   mVertices[j + i * mHiResVerticesPerSide].Set_Y(
      mDownTile->GetVertex(mHiResVerticesPerSide - 1, 0).Get_Y()
      );
}
// Lower border AND with a down neighbour
else if(i == 0 && j > 0 && mDownTile != NULL)
{
   mVertices[j + i * mHiResVerticesPerSide].Set_Y(
      mDownTile->GetVertex(mHiResVerticesPerSide - 1, j).Get_Y()
      );
}
// Left border AND with a left neighbour
else if(i > 0 && j == 0 && mLeftTile != NULL)
{
   mVertices[j + i * mHiResVerticesPerSide].Set_Y(
      mLeftTile->GetVertex(i, mHiResVerticesPerSide - 1).Get_Y()
      );
}
// Other vertices
else
{
   if(i < (mHiResVerticesPerSide >> 1))
   {
      if(j <= i)
      {
         mVertices[j + i * mHiResVerticesPerSide].Set_Y(
            (GLfloat)j * mPyramidGradient
            -(GLfloat)(pNoize) + (GLfloat)(rand() % (2 * pNoize + 1))
            );
      }
      else if(j >= (mHiResVerticesPerSide - 1) - i)
      {
         mVertices[j + i * mHiResVerticesPerSide].Set_Y(
            (GLfloat)(mHiResVerticesPerSide - 1 - j) * mPyramidGradient
            -(GLfloat)(pNoize) + (GLfloat)(rand() % (2 * pNoize + 1))
            );
      }
      else
      {
         mVertices[j + i * mHiResVerticesPerSide].Set_Y(
            (GLfloat)i * mPyramidGradient
            -(GLfloat)(pNoize) + (GLfloat)(rand() % (2 * pNoize + 1))
            );
      }
   }
   else
   {
      if(j < mHiResVerticesPerSide - i)
      {
```

```
                        mVertices[j + i * mHiResVerticesPerSide].Set_Y(
                           (GLfloat)j * mPyramidGradient
                           -(GLfloat)(pNoize) + (GLfloat)(rand() % (2 * pNoize + 1))
                           );
                     }
                     else if(j >= i)
                     {
                        mVertices[j + i * mHiResVerticesPerSide].Set_Y(
                           (GLfloat)(mHiResVerticesPerSide - 1 - j) * mPyramidGradient
                           -(GLfloat)(pNoize) + (GLfloat)(rand() % (2 * pNoize + 1))
                           );
                     }
                     else
                     {
                        mVertices[j + i * mHiResVerticesPerSide].Set_Y(
                           (GLfloat)(mHiResVerticesPerSide - i - 1) * mPyramidGradient
                           -(GLfloat)(pNoize) + (GLfloat)(rand() % (2 * pNoize + 1))
                           );
                     }
                  }
               }
            }
            else if(k == 2) // Z
            {
               mVertices[j + i * mHiResVerticesPerSide].Set_Z((GLfloat)i  /
               (GLfloat)(mHiResVerticesPerSide - 1) * (GLfloat)mTileSize);
            }
         }
      }
   }
}
```